EPFL

Thèse n° 7290

Squall: Scalable Real-time Analytics using Efficient, Skew-resilient Join Operators

Présentée le 15 mai 2023

Faculté informatique et communications
Laboratoire de théorie et applications d'analyse de données
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Aleksandar VITOROVIĆ

Acceptée sur proposition du jury

Prof. W. Zwaenepoel, président du jury
Prof. C. Koch, directeur de thèse
Prof. V. Markl, rapporteur
Dr S. Kandula, rapporteur
Prof. A. Ailamaki, rapporteuse

École polytechnique fédérale de Lausanne

2023

I am fain to compare myself with a wanderer
on the mountains who, not knowing the path,
climbs slowly and painfully upwards
and often has to retrace his steps
because he can go no further — then,
whether by taking thought or from luck,
discovers a new track that leads him
on a little till at length
when he reaches the summit
he finds to his shame
that there is a royal road
by which he might have ascended,
had he only the wits to find
the right approach to it. In my works,
I naturally said nothing about my mistake
to the reader, but only described
the made track by which he may now reach
the same heights without difficulty.
— Hermann von Helmholtz

To my beloved wife, Milica,
and to my dear daughter, Sara.

Acknowledgements

First of all, I would like to thank my advisor, Christoph Koch. When I came, I was completely clueless about what the research is all about, and he was patient in explaining me how to do it the right way. Christoph taught me how to distinguish between incremental and fundamental advances in science, and to think big. I appreciate that he was always asking more from me, forcing me to give my best. He also showed me by example a high level of dedication a research career requires. I am also thankful to him that he almost always pulled an all-nighter with us before a deadline. I would also like to thank Srikanth Kandula, my advisor at Microsoft Research, where I interned during summer 2015. Thank you for providing me with such a great environment and for assembling such a great team of highly qualified and motivated people. It was my pleasure working and brainstorming with you! I learned a lot from you Srikanth, especially in terms of getting great ideas and getting them done in a real system.

I owe many thanks to Yannis Klonatos, my dear colleague from the EPFL DATA Lab. He helped me in so many different ways, so that it is hard to express in words all the gratitude I feel towards him. Several times during my PhD when I was close to quiting because of slow progress, he gave me feedback about my work, and encouraged me to continue. He reviewed every paper that I ever submitted, and not once, but many times. Thank you Yannis for countless proof-readings of my papers! In particular, I thank you for your enormous contributions to the ICDE paper. You read the paper in great detail (including all the formulas), and helped me to extract and articulate contributions in a clear way. I am sure that if you were not there, this paper would not be accepted, and I would have hard time to graduate.

I thank Mohammad El Seidy for being a core contributor of Squall (this thesis is about our system Squall). Thank you for the collaboration on the papers, and all the brainstorming that helped us to better design Squall. I'll never forget these two weeks before an important paper deadline when you stopped your own research to help me to submit a paper.

I also highly appreciate the contributions of many master students, whose semester, course or master projects extended Squall and made it much more robust. Most notably, I thank Khayyam Guliyev and Khue Vu. Khayyam created a Web interface for Squall, and implemented some multi-way join operators. Khue enabled running Local DBToaster from Squall, and led a team in a course project that implemented a hypercube scheme. I also thank all the other Squall contributors: Mohammed El Seidy and Christoph Koch (core contributors), Yannis Klonatos, Abdallah Elguindy, Oliver Kennedy, Amir Shaikhha, Mohammad Dashti, Daniel Espino Timón, Tam Nguyen Thanh, Diana-Andreea Popescu, Zisi Wang, Loïc Gardiol, Bruno Corijn, Yannick Tapparel, Michalis Zervos, Matthaios-Alexandros Olma, Andriani Stylianou,

Guillaume Ulrich, Patrice Gueniat, Gilles Cressier, Romain Poiffaut, Nithin George and Ferhat Elmas.

I would like to thank the entire DATA lab. I thank Mohammed El Seidy for listening to me when I was complaining, and for countless discussions about the research and industry, work-life balance and true values in life, faith, philosophy, to name just a few. You definitely helped me to get a wider perspective on life. Thank you, Miloš Nikolić, for being a great office-mate during last 5 years, and Lionel Parreaux, with whom we shared the office for a semester. I thank Amir Shaikhha and Mohammed Dashti for always having time to discuss both professional and non-professional topics, and for radiating optimism even in difficult situations. On the other hand, I appreciate honesty of Daniel Lupei, whose back-to-earth comments were mostly true. Daniel, I'll also remember our runs on the shore of Lake Geneva, and your patience until I got into good shape. I thank Immanuel Trummer for his in-depth feedback on my ICDE paper, and for the translation of the abstract of this thesis to German. I thank Simone Muller, for helping out in many administrative tasks. I also thank Matthaios Olma from the DIAS Lab, with whom I had a great collaboration at Microsoft Research.

No success is possible without a strong family support. First, I owe greatest thanks to my wife Milica for all the love, care, encouragement and understanding. She made my life much more enjoyable. I also thank her for tolerating my long working hours before the deadlines and for patiently listening about research in computer science, despite the fact that she is a professor of music. I thank our daughter Sara, who always run to me and hugged me when I was coming back from work. Frequently I worked at home, and she was full of understanding and encouraged me with "Aaa, pos'o!" ("Ahh, work!") and "Tata piše tezu" ("Daddy writes his thesis!"). I thank my parents, Rade and Zora, for their unconditional love and moral and financial support throughout whole my life. They taught me what are the core values in life, and to work hard. I thank to my brother Danilo, who taught me how to write my first Hello-world program in Basic on Commodore 64 when I was around 10. He is now an assistant professor of neurological sciences, but he knows enough about computer science so he was able to give me some feedback about my research. Last but not least, I thank my mother-in-law Mira for coming to Switzerland many times to help us with Sara.

I would also like to thank to many of my friends I got to know here in Switzerland. I thank Clifford and Samuel for encouraging me during my studies. I also thank Stephane and Marco, who invited us many times at their place. I am thankful to a Serbian EPFL community (Miloš, Milena, Andrej, Lazar, Bilja, Zlatko, Nataša, Renata, Ivan, Danica, Mira) for many lunches and birthday parties we had. Thank you, Vlad, for many interesting discussion we had, and for tutoring me on how to find a job. I would also like to thank Quentin (whom I met at Mission Caleb) for translating the abstract of this thesis in French. I thank Groupes Bibliques Universitaires (GBU) at EPFL, for organizing many interesting events and conferences.

Finally, I would like to thank Dositeja, Foundation for young talents of Republic of Serbia, for granting me a scholarship during my PhD studies.

Lausanne, 15 September 2016 A. V.

Abstract

Squall is a scalable online query engine that runs complex analytics in a cluster using skew-resilient, adaptive operators. Online processing implies that results are incrementally built as the input arrives, and it is ubiquitous for many applications such as algorithmic trading, clickstream analysis and business intelligence (e.g., in order to reach a potential customer during the active session).

This thesis presents an overview of Squall, including some novel join operators, as well as lessons learned over five years of working on this system. Existing open-source online systems (e.g. Twitter Storm, Spark Streaming) provide only hash-joins, which are limited to equi-joins and prone to skew. In contrast, Squall puts together state-of-the-art skew-resilient partitioning schemes (including some of our own), local query operators, and techniques for scalable online query processing. Such a system allows us to leverage the effect of various design choices on the performance, to seamlessly build efficient novel operators, and to discover and address new skew types (e.g. dependence on tuple arrival order) that can arise only in online systems.

Existing partitioning schemes for joins work well only for a narrow set of data distribution properties, that is, specific proportion of join output and input sizes for 2-way joins, or similar data distribution among all the relations for multi-way joins. In contrast, Squall covers the entire spectrum of different data distributions by providing two novel skew-resilient partitioning schemes: (a) a scheme for 2-way non-equi joins partitions the data using a multi-stage load-balancing algorithm that contains a join-specialized computational geometry algorithm, and (b) a scheme for multi-way joins which constructs a composite partitioning, consisting of different partitioning schemes according to the skew degree in different relation attributes. Compared to state-of-the art, our schemes achieve up to 12× speedup and are up to 5× more efficient in terms of resource consumption.

Key words: online query engine, skew-resilient parallel operators, 2-way and multi-way joins, equi and non-equi joins, skew types, adaptivity

Résumé

Squall est un moteur de réquete évolutif en ligne qui exécute des analyses complexes dans un cluster en utilisant des opérateurs adaptatifs et asymétrie-résilients. Le traitement en ligne implique que les résultats sont incrémentiellement construits dès que l'entrée arrive, ce qui est très souvent le cas pour un grand nombre d'applications comme le trading algorithmique, l'analyse de flux de clics et la informatique décisionnelle (par exemple afin d'atteindre un client potentiel au cours d'une session active).

Cette thèse présente une vue d'ensemble de Squall, comprenant plusieurs opérateurs de jointure, ainsi que quelques leçons retenues après cinq années de travail sur ce système. Les systèmes open-source en ligne existants (comme Twitter Storm et Spark Streaming) fournissent seulement des jointures hachée, qui sont limités aux equi-jointures et sujets aux asymétrie. En revanche, Squall utilise à la fois des schémas de partitions asymétrie-résilients de pointe (y compris certains des notres), des opérateurs de requêtes locales, et des techniques pour le traitement évolutif de requêtes en ligne. Un tel système permet d'évaluer l'effet de différents choix de conception sur la performance, de construire de manière transparente de nouveaux opérateurs efficaces, et de découvrir et solutionner des nouveaux types de asymétrie (par exemple dépendance au tuple ordre d'arrivée) qui ne peut survenir que dans des systèmes en ligne.

Les schémas de partitions existants pour les jointures fonctionnent bien seulement pour un ensemble restreint de propriétés de distribution de données, donc une proportion spécifique de taille entre de entrée et sortie de jointure pour des jointures 2-voies, ou la distribution de données similaire parmi toutes les relations pour des jointures multi-voies. En revanche, Squall couvre tout le spectre des différentes distributions de données en fournissant deux nouveaux schémas de partitions asymétrie-résilients : (a) un schéma pour 2-voies non-equi jointures qui divise les données en utilisant un algorithme d'équilibrage multi-étage qui contient un algorithme de géométrie algorithmique spécialisée pour jointures et (b) un schéma pour des jointures multi-voies qui construit un partitionnement composite, constitué de différents schémas de partition en fonction du degré de asymétrie dans les différents attributs des relations. Par rapport aux schémas actuels de pointe, nos schémas atteignent une vitesse jusqu'à 12 fois supérieure et sont jusqu'à 5 fois plus efficace en termes de consommation de ressources.

Mots clefs : Moteur de réquete en ligne, opérateurs parallèles asymétrie-résilients, jointures 2-voies et multi-voies, equi et non-equi jointures, types d'asymétrie, adaptivité

Zusammenfassung

Squall ist ein skalierbares System zur Online-Datenverarbeitung welches komplexe Analysen auf einem Cluster ausführt. Dafür verwendet Squall verzerrungsresistente und anpassungsfähige Operatoren. Squall zielt ab auf Online-Szenarien in denen Eingabedaten inkrementell verfügbar werden. Solche Szenarien treten zum Beispiel auf im Kontext des Hochfrequenzhandels, der Clickstream-Analyse, oder im Bereich der Geschäftsanalyse.
Diese Dissertation beschreibt das Squall System, seine neuartigen Join Algorithmen und die praktischen Erfahrungen die wir über die letzten Jahre im Umgang mit dem System gesammelt haben. Frühere Systeme (zum Beispiel Twitter Storm order Spark Streaming) implementieren nur den Hash Join Algorithmus welcher auf Gleichheitspraedikate beschränkt ist und empfindlich auf Verzerrungen reagiert. Squall setzt sich ab von diesen früheren Systemen durch verzerrungsresistente Datenverteilungsmethoden, lokale Datenverarbeitungsoperatoren und durch Techniken zur skalierbaren Online-Verarbeitung. Ein solches System erlaubt es uns, die positiven Effekte von verschiedensten Design-Entscheidungen miteinander zu kombinieren. Frühere Datenverteilungsmethoden decken nur einen kleinen Teil der möglichen Verzerrungen ab. Squall deckt dagegen das gesamte Spektrum verschiedener Fälle ab und bietet zwei neue verzerrungsresistente Datenverteilungsmethoden an: (i) eine Datenverteilungsmethode für binäre Joins mit diversen Filterprädikaten und (ii) eine Datenverteilungsmethode für nicht-binäre Joins. Verglichen mit dem vorherigen Stand der Technik beschleunigen unsere Methoden die Verarbeitung um Faktor 12 und reduzieren den Ressourcenverbrauch um Faktor 5.

Stichwörter: Online Datenverarbeitungssystem, verzerrungsresistente Operatoren zur parallelen Verarbeitung, binäre und nicht-binäre Joins, Joins mit diversen Filterprädikaten, Verzerrungsarten, Anpassungsfähigkeit

Contents

Acknowledgements **i**

Abstract (English/Français/Deutsch) **iii**

List of figures **xiii**

List of tables **xv**

1 Introduction **1**
1.1 Thesis statement . . . 1
1.2 Motivation . . . 3
1.3 Intellectual and technological contributions . . . 3
1.4 Long and short-term impact . . . 5
1.5 Thesis outline . . . 5

2 Background **7**
2.1 Classes of online processing . . . 8
2.2 Requirements for online systems . . . 10
2.3 Existing work on online systems . . . 12

3 System architecture **17**
3.1 Overview . . . 17
3.2 Consistency . . . 23

4 A partitioning scheme for 2-way Joins **27**
4.1 Introduction . . . 27
4.2 Background & Preliminaries . . . 30
4.2.1 Definitions . . . 30
4.2.2 Content-Insensitive Partitioning Scheme . . . 32
4.2.3 Content-Sensitive Partitioning Scheme . . . 33
4.2.4 Equi-Weight Histogram Scheme . . . 34
4.3 Histogram algorithm . . . 35
4.3.1 Sampling . . . 38
4.3.2 Coarsening . . . 39
4.3.3 Regionalization . . . 40

4.3.4 Putting it all together . . . 43
4.4 Join operator . . . 44
4.4.1 Sampling the Output Tuples . . . 44
4.4.2 Discussion and Generalization . . . 46
4.5 Related Work . . . 46
4.6 Evaluation . . . 49
4.6.1 Experimental Setup . . . 49
4.6.2 Performance Analysis . . . 51
4.6.3 Scalability . . . 53
4.6.4 Accuracy and Efficiency of CS_{IO} . . . 55
4.6.5 Sensitivity analysis . . . 56
4.6.6 Summary . . . 60
4.7 Further details . . . 60
4.7.1 Types of partitioning . . . 60
4.7.2 The histogram algorithm: Details and proofs . . . 61
4.7.3 Joins . . . 65

5 Multi-way join operators: partitioning schemes and local operators 67
5.1 Novel join operators . . . 67
5.1.1 Applications . . . 67
5.1.2 Partitioning schemes . . . 69
5.1.3 Important special cases . . . 72
5.1.4 Local join algorithms . . . 72
5.1.5 HyLD operator: Hypercube scheme with Local DBToaster . . . 73
5.2 Multi-way joins: General case . . . 74
5.3 Gathering insights about multi-way joins . . . 78
5.3.1 The subsystem for collecting results and performance metrics . . . 79
5.3.2 Interacting with the system . . . 81
5.4 Related work . . . 82
5.5 Evaluation . . . 85
5.5.1 Datasets . . . 86
5.5.2 Multi-way vs 2-way joins . . . 86
5.5.3 Hybrid-Hypercube versus Hash-Hypercube and Random-Hypercube . . . 87
5.5.4 DBToaster versus traditional local joins . . . 91
5.5.5 Summary . . . 92

6 Adaptivity 93
6.1 Skew types and Adaptivity . . . 93
6.2 Adaptive 1-Bucket operator . . . 96
6.2.1 Operator Structure . . . 97
6.2.2 Input-load factor . . . 98
6.2.3 Adaptivity . . . 99
6.3 Towards adaptive Equi-weight histogram (EWH) scheme . . . 102

6.3.1 Monitoring Statistics . . . 102
6.3.2 Actuation . . . 103
6.4 Evaluation for Adaptive 1-Bucket . . . 104
6.4.1 Skew Resilience . . . 106
6.4.2 Performance Evaluation . . . 106
6.4.3 Scalability Results . . . 109
6.4.4 Summary . . . 110

7 Conclusion **111**
7.1 Summary of Contributions . . . 111
7.2 Future work . . . 112

A Appendix **115**
A.1 Integrating DBToaster in Squall . . . 115

Bibliography **126**

Curriculum Vitae **127**

List of Figures

3.1 Squall architecture. An example query plan has selections (σ), projections (π), joins ($\bowtie$) and aggregations (Agg). 19

4.1 Different partitioning schemes (of 3 machines) on a band-join with a join condition $|R_1.A - R_2.A| \leq 1$. Shaded cells represent output tuples. (b)-(d) I_r is $input$ and O_r is $output$ metric of a region r with maximum weight $w_{x \in 1..J} = I_x + O_x$. 32
4.2 Weight Histograms. 35
4.3 Histogram algorithm stages. The weight function is $w(r) = c_i(r) + c_o(r) = input(r) + output(r)$. For instance, in (c), $w(r_4) = 2 \cdot 112 + 15 = 239$. 37
4.4 a) Non-monotonicity in rectangles 1 and 2 due to candidates marked with black. b) r_{m1} (r_{m2}) is a minimal candidate rectangle for r_1 (r_2). 40
4.5 Execution times 51
4.6 Cluster Memory Consumption. 53
4.7 Scalability of B_{CB-3}. 54
4.8 Scalability of BE_{OCD}. 54
4.9 Maximum Region Weight. 55
4.10 Types of partitionings. 60

5.1 Partitioning schemes for $R(x,y) \bowtie S(y,z) \bowtie T(z,t)$. Uniform data (a), data-independent (b), skewed data (c, d). 70
5.2 Squall's Web Interface. 79
5.3 Demonstration: Running a query. 80
5.4 Results and query performance metrics. 80
5.5 Finding bottleneck in a Squall query plan. sel stands for a no-op selection (it passes through all the tuples). 82
5.6 Performance for 3-reachability query. We use 36 joiner machines. 87
5.7 Comparison of different hypercube schemes. 88
5.8 Multi-way joins with different local joins (traditional vs DBToaster). 92

6.1 The adaptive operator structure. Each of J machines is assigned one reshuffler and one joiner task. 97
6.2 (I) Relations are of the same size (a), so 8×8 is the optimum mapping (b). (II) Both relations grow, and one relation is 64× bigger than the other one (c). Using the old 8×8 mapping (d) is highly suboptimal compared to (64×1)-mapping (e). 98

6.3 Decomposing $J = 20$ machines into independent groups of 16 and 4 machines. 101
6.4 Input-Load Factor. 106
6.5 Execution Time. 107
6.6 Throughput and latency. 108
6.7 B_{NCI} Performance Evaluation. 108
6.8 Scalability Results. 109

List of Tables

4.1 Comparison with most important related work. 30
4.2 Summary of the notation used in the paper. 31
4.3 The time complexity improvements. 36
4.4 Joins' characteristics. $Input$ and $output$ sizes are in millions of tuples. β is the width of the band. 50
4.5 Join execution and histogram algorithm time (s) of CS_I for different number of buckets p. 52

5.1 Relation and final result schemas for the Stock market query. 68
5.2 Maximum and average load per machine for different hypercube schemes. M stands for millions of tuples. 90
5.3 Replication factor for different hypercube schemes. 90

6.1 Runtime in secs. 105

1 Introduction

1.1 Thesis statement

We design and implement Squall, a scalable online query engine that runs complex analytics in a cluster. First, we clarify what we mean by "online". Online processing implies that results are incrementally built as the input arrives. A tuple is a data unit that contains one or more data type objects, and is sometimes referred to in the literature as a record. Each input tuple produces output and updates the system state necessary for processing subsequent inputs. Online processing is ubiquitous for many applications such as algorithmic trading, clickstream analysis and business intelligence (e.g., in order to reach a potential customer during the active session).

Scalable online processing is challenging in the presence of data skew [134]. Skew is a tuple and key distribution that leads to uneven data partitioning. Skew occurs in real-world datasets and applications [140, 36, 20]. In the context of scientific MapReduce [48] jobs, [115] shows that assigning the same input data size to reducers is insufficient for load-balancing. Namely, the authors show that 38% of Hadoop [i] jobs in a cluster running scientific applications suffer from considerable skew in the amount of work per reducer (there exists a reducer task which takes at least 2X time that of an average task). On the other hand, join processing takes a central place in many analytics tasks. We distinguish two types of skew in join operators. Redistribution skew (RS) represents uneven input data partitioning among the machines [ii] due to skew in the join keys. Join product skew (JPS) represents imbalance in load due to variability in the join selectivity. In both cases, a small number of machines process most of the data. Squall addresses both RS and JPS.

Squall achieves high throughput and low latency using skew-resilient, scalable and adaptive operators. We study database operators such as selections, projections, joins and aggregations, but we focus on joins, as they are the most challenging ones. Squall supports both 2-way

[i] Hadoop is an open-source implementation of the MapReduce.

[ii] By machine we mean a logical concept of a computing node, rather than a physical, potentially multi-core machine.

joins and multi-way joins. By a multi-way join we mean a join between multiple relations that requires a single communication step (in MapReduce [48] terminology, one MapReduce job). Each operator runs on multiple machines (by machine we mean a core/hardware thread with exclusively assigned portion of the main memory). To provide for scalability even in the case of high number of machines, we employ shared-nothing architecture. In this setting, an operator consists of a partitioning scheme and a local operator. A partitioning scheme assigns incoming tuples to the operators' machines (with or without replication). A local operator runs the same algorithm on each of the machines, but on different data (data parallelism). The role of a partitioning scheme is to evenly partition the data among the machines, that is, to achieve load balancing.

Squall builds on state-of-the-art partitioning schemes and local algorithms, including some of our own. As we already said, we focus on joins. An ideal partitioning scheme is skew resilient, that is, it achieves load balancing despite possible skew in the data. By load balancing we imply minimizing the maximum work per machine. Previous work achieves this goal and offers efficient solutions only in some situations. For 2-way joins, existing approaches work well only for a specific proportion between the join output and input sizes. They also require that this proportion is known beforehand. Unfortunately, output size estimation techniques are known to be error-prone [74]. For multi-way joins, previous work is designed only for the cases when a) all the joins are equi-joins, and skew exists either in all the join keys or in none of them, or b) all the joins are non-equi joins. Consequently, the existing work falls short for a common case of datasets that mix uniform distribution for some relation attributes and skewed distribution for other attributes, or for queries that consist of both equi- and non-equi joins. In contrast, Squall covers the entire spectrum of different data distributions. Namely, we design and implement a partitioning scheme for 2-way non-equi joins, which collects detailed information about both input and output data distribution. Using this information, the scheme optimally partitions the input data among the machines. For multi-way joins, we propose a "composite" (multi-dimensional) partitioning scheme, which consists of different "atomic" (per-attribute) partitioning schemes. We build a composite scheme according to the skew degree in different relation attributes. Last but not least, our partitioning schemes are applicable in an offline system as well.

Furthermore, Squall offers state-of-the-art local join operators, including DBToaster [16]. DBToaster was previously considered hard to parallelize in the presence of multiple (parallel) data sources [79], but Squall provides a natural framework for its parallelization.

In the context of online processing, we discuss how Squall operators can adapt to changes in data statistics. Statistics in an online system may be unknown ahead of time, or it changes during run-time. On the other hand, we need data statistics to choose an optimal partitioning. We design and implement an adaptive 2-way join operator that has optimality guarantees on data distribution and communication costs. Existing open-source online systems (e.g., Twitter's Storm [95], Spark Streaming [143], Flink [22] do not provide adaptive operators. On the other hand, existing adaptive partitioning schemes do not provide optimality guarantees

as Squall does. We also explain that merely adjusting to data statistics is sometimes insufficient to achieve good performance. An example is skew fluctuation, which implies changing data statistics right after the operator adapts.

1.2 Motivation

Skew occurs frequently in real-life datasets. For instance, certain types of skewed distributions (such as zipfian distribution) appear in Internet packet traces, city sizes and word frequency in natural languages. Unfortunately, existing open-source online systems (e.g., Twitter's Storm [95], Spark Streaming [143], Flink [22][iii]) provide only vanilla database operators, such as equi-joins based on hash partitioning, which do not perform well in the case of skew [iv]. Regarding non-equi joins, Storm do not provide them. Whereas, Spark Streaming and Flink execute non-equi joins very inefficiently (a Cartesian product followed by a selection). On the other hand, existing partitioning schemes (both for equi- and non-equi joins) work well only for a narrow set of data distribution properties. Squall addresses this problem. It allows studying different partitioning schemes, local query operators and techniques for scalable online query processing in a unified framework.

1.3 Intellectual and technological contributions

1. Partitioning scheme for 2-way joins. Our main contribution is a skew-resilient scheme for 2-way non-equi joins. Compared to state-of-the art, it achieves sizable gains under a wide variety of conditions. Our scheme achieves *minimal* work per machine, without imposing any assumptions about input or output sizes, or data distribution. To do so, we devise an efficient parallel scheme for capturing the input and output distribution from the join to a special data structure called join matrix. Each dimension of the join matrix (rows and columns) corresponds to the join keys from an input relation, and the join matrix contains the information on work distribution for processing the corresponding ranges of join keys. In particular, we build the join matrix through sampling of both input and output data, without performing the entire join. To optimally partition the work (join matrix) among the machines, we devise a multi-stage load-balancing algorithm which contains a novel, join-specialized computational geometry algorithm for rectangle tiling.

2. Partitioning scheme for multi-way joins. The second contribution is our partitioning scheme for multi-way joins. We designed and implemented a partitioning scheme that is a composite of multiple per-attribute partitioning schemes. We choose appropriate per-attribute partitioning scheme according to the data distribution on the corresponding relation's attribute (whether there is skew or not) and the attribute's connections to other relations' attributes through join conditions. It was challenging to design an optimization algorithm

[iii] Flink provides both offline and online processing, but in this thesis we discuss only the online case.

[iv] Some of these systems also provide a range partitioning, which can address RS, but it is still prone to JPS.

(an algorithm that generates on optimal composite partitioning scheme) given the variety of join conditions and variety of different data distributions. We show that the optimization algorithm is an elegant extension of an existing algorithm (which is already proven optimal).

3. Adaptive operators. Our adaptive operator follows general adaptivity loop [50] that contains a) capturing the statistics from the past, b) deciding on changing the partitioning scheme and c) migrating the data accordingly. Our contributions are the following. First, we collect statistics in a decentralized manner (rather than collecting all the statistics on a single machine). Second, we carefully choose when to re-evaluate the optimality of the current partitioning scheme (and accordingly adjust the scheme). We prove a constant competitive ratio in data distribution optimality, as well as amortized total communication cost (including migrations). Third, we perform minimal, non-blocking state migration by reusing existing state on the machines as much as possible.

4. Modular design and skew in online systems. Finally, we design Squall so that it puts together state-of-the-art partitioning schemes, local query operators, and techniques for scalable online query processing. Such a system allows us to leverage the effect of various design choices on the performance, to seamlessly build efficient novel operators, and to think about new aspects, such as dependence on tuple arrival order and support for fault-tolerance. We discover new types of skew, that can arise only in online systems. For instance, temporal skew occurs even if the data distribution is uniform, if there is a specific tuple arrival pattern (e.g., tuples arrive in the sorted order and the tuple key frequency is moderate). Squall offers techniques to address these types of skew. We also classify partitioning schemes according to levels of adaptivity that they achieve for different skew types. This study leads us to discover and formulate a general principle about tradeoffs between **S**kew-resilience, **A**daptivity and **R**eplication (SAR principle).

These contributions and this thesis are derived from the publications accepted at VLDB and ICDE conferences:

- Aleksandar Vitorovic, Mohammed Elseidy, Khayyam Guliyev, Khue Vu Minh, Daniel Espino, Mohammad Dashti, Ioannis Klonatos, Christoph Koch.
  Squall: Scalable Real-time Analytics.
  VLDB Demo 2016.

- Aleksandar Vitorovic, Mohammed Elseidy, Christoph Koch.
  Load Balancing and Skew Resilience for Parallel Joins.
  ICDE 2016.

- Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, Christoph Koch.
  Scalable and Adaptive Online Joins.
  VLDB 2014.

For the last paper, I have to thank Mohammed and Abdallah for their contributions. My

contributions for this paper were:

- This paper originated from a course project for which I was the responsible TA. The course project was about implementing a skew-resilient join operator in Squall, while Mohammed introduced adaptivity in this operator.
- I participated in the adaptive operator design. Most notably, I was responsible for the aspects of fault-tolerance and correctness in the case of multiple operator groups. The groups are necessary when the number of machines is not a power of 2. In that case, we divide the operators into groups such that each group has a power of two machines.
- I was responsible for running the experiments, and writing the Evaluation section. This was quite challenging, as it required tuning Squall and Storm to harness the best performance.

1.4 Long and short-term impact

First, Squall is an open-source project[v] that has been developed for the last five years (mainly by the authors at EPFL, but also with external contributions). It has been available for several years, and it has attracted a community of users. We anticipate that the number of users will grow due to the following. As existing online systems do not handle skew well, users will employ Squall whenever there is a need for skew-resilient operators. In contrast to these systems, Squall pays attention to non-equi joins, which are ubiquitous nowadays for expressing complex analytics tasks. Second, Squall is a modular system and it can be easily extended with new partitioning schemes and local operators. This might attract scientist to implement their operators in Squall. For instance, they might want to evaluate their local join along with some of our partitioning schemes. Third, our partitioning scheme for 2-way joins may change the way scientists look at join processing. Previously, scientists tried to reduce the time for collecting data statistics. Our work shows that it is worth spending more time for capturing the data distribution, as it pays off in the resulting optimal partitioning schemes. Finally, our multi-way join can significantly improve the performance, and the optimization algorithm is very elegant. Our partitioning schemes are also applicable in offline scenarios, so we expect commercial systems to implement it.

1.5 Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 gives more details about background. It explains motivation for our work, including the emergent requirements for systems that provide online processing and skew resilience. Chapter 3 presents the high-level architecture of our system Squall. In this chapter, we highlight supported features of the system, as well as process of translating queries to the execution graph in a distributed

[v] https://github.com/epfldata/squall/

setting. Chapter 4 introduces a novel partitioning scheme for 2-way joins, including a detailed theoretical analysis of the scheme. This section also provide a detailed overview of state-of-the-art 2-way join partitioning schemes, as well as a comparison with the most relevant state-of-the-art techniques. Chapter 5 introduces multi-way joins in Squall. After explaining the motivation for multi-way joins both for offline and online systems, this chapter presents a novel partitioning scheme for multi-way joins. We also illustrate modularity of Squall by showing how we wire up our partitioning scheme with a state-of-the-art local join operators. Chapter 6 describes different types of skew, some of which exist only in online systems. We categorize different partitioning schemes according to their resilience to these types of skew. In this chapter we also introduce the SAR principle, which represents a tradeoff between **S**kew-resilience, **A**daptivity and **R**eplication. Furthermore, this chapter presents an adaptive operator design for a 2-way join, which can be generalized to multi-way joins. Finally, Chapter 7 concludes this thesis.

2 Background

There is an ever-increasing need for processing large data in a scalable way. To get useful insights from the data, we need process terabytes of data such as logs or clickstream data. Another interesting use case is exploratory data analysis over scientific data. An example is analyzing particle physics experiments at CERN, where each year scientists analyze 30 petabytes of data collected from particle accelerator Large Hadron Collider (LHC) [3]. In addition, a petabyte of data from LHC is ingested and analyzed on daily basis.

In fact, volume of the data is just one requirement in data analytics. Analytics tools also need to provide for high velocity, which refers to speed of data changes [i]. The system needs to quickly (and continuously) adjust the result according to these data changes. Here is a list of common applications that require continuous and quick answers:

- **Business intelligence.** It is crucial to find patterns in customer and sales data, in order to reach potential customers (e.g., offer them promotions) during an active session. To do so, we need the information about customers' salary, previous history of shopping etc. Finding potential customers typically involves joining customer, product and session tables. For example, Amazon offers product recommendations according to the session information, including last visited web page [1]. At Twitter, the recommendations are based on recent conversations [81]. Many start-ups, such as QuantCast, 8digits and RocketFule, created their businesses around the idea of facilitating online advertising.
- **Online anomaly and fraud detection.** For instance, it is crucial to perform fraud detection over credit card transactions quickly and continuously, in order to reduce the financial loss due to credit card misuse. Running fraud detection algorithms with low latency is also necessary for virtual auction systems, such as eBay and BetFair, where we need to continuously analyze trading transactions.
- **Stock market and algorithmic trading.** Bidding (and accepting bids) on stock market also require fast and continuous processing. In algorithmic trading, arbitrage is of great

[i] This includes updates to existing data as well as new data

interest, as it allows earning money by selling goods on one market and buying them on another market with lower price. Arbitrage not only requires low latency (in order to be the first to perform a trade), but it also implies complex join processing (we discuss the existing work on complex join processing in Section 2.3). Trading systems can also perform analytics over external data providers, such as social networks, and use this information to improve its trading strategies. For instance, if there is a lot of positive hype around a company (which we can for example obtain by running a sentiment analysis tool over Tweeter's tweets), it is likely that its share price will increase. Joining trading and social media data typically involves complex query processing, and Squall is designed for that.

- **Monitoring.** For safety reasons, it is crucial that infrastructure surveillance and traffic monitoring provide low-latency processing. In other domains, such as sensor network and Internet of Things, we still need to analyze incoming data in a real-time fashion in order to obtain actionable insights.

The common denominator in all these applications is a quick reaction on changes (new data tuples coming to the system), which lends itself to emergence of *online* analytics. In some of these cases, we need to ensure latency (the timeout between the time a tuple comes and the time when the tuple is reflected in the result) on the order of tens or hundreds of milliseconds.

Next, we discuss classes of query processing and their different requirements. Offline systems include traditional Relational Database Management Systems (RDBMS) such as Oracle Database and Microsoft SQL Server, and MapReduce systems such as Hadoop [7] and Cosmos/SCOPE [38]. All these systems perform arbitrary queries over a static dataset. However, completing a query may take hours or even days. On-line analytical processing (OLAP) systems provide faster answers than offline systems for a set of queries (e.g., per-dimension aggregations over multi-dimensional data). They do so by materializing views and computing some aggregations ahead of time. In OLAP systems, datasets are mostly static, possibly with periodic (e.g., nightly) updates. Furthermore, the answer is provided only after all the data (materialized views and base relation) that contributes to the result is fully processed. In contrast, online processing implies that updates (tuples) are continuously coming to the system at high pace. Online systems need to provide low latency for processing incoming tuples and producing the final result. To do that, these systems materialize some views, similarly to OLAP systems. Squall is a system that supports large-scale online (sometimes also called real-time) query processing.

2.1 Classes of online processing

There are three main classes of online processing: incremental view maintenance, stream processing and online aggregation.

To begin with, *incremental view maintenance* (IVM) [116, 69, 141, 67, 80] represents the query

result as a view and it continuously updates the view as tuples are coming to the system. The goal is to maintain the view without re-computation (performing the full query from scratch) each time a new tuples arrives. In addition to maintaining the view, an online system may continuously send a view (result) deltas to downstream operators (for further processing) or to a completely different subsystem (e.g., to a visualization tool such as Graphite [5]). Squall supports classical IVM as well as these extensions.

Stream processing refers to processing very large (potentially unbounded) streams of data using limited amount of resources (memory). There are several flavors of stream processing, and window semantics is the most popular one. Window semantics (cf. e.g. [78]) implies that we maintain and perform operations only on the state consisting of tuples that recently arrived to the system (e.g. tuples from time/count-based tumbling or sliding windows). Alternatives are to perform load shedding which implies discarding some incoming tuples (e.g. [124]), or to preserve a synopses (approximated representation) of the entire state [47]. Existing streaming systems such as Borealis [10] and STREAM [23] focus on small-state windows and load shedding. These systems partition the operators among the machines, providing inter-operator parallelism. In contrast, Squall schedules operators using large-scale intra-parallelism in addition to inter-parallelism. This is necessary as Squall is designed for large-state operators. Squall does not implement load shedding, but it supports window join semantics by reusing the machinery for IVM and extending it with constructs for removing outdated tuples from the operator state.

Finally, *online aggregation* (e.g., [77]) focuses on the case of aggregate queries, and produces an approximate query result, long before the processing of the query has been completed. The dataset is static and known ahead of time. Using the amount of data processed so far, online aggregation also provides confidence intervals and error bounds on the approximate result. That is, it gives the information about the confidence that the approximate result is within certain error bounds from the exact answer (the exact answer is the answer on the entire dataset). As more data is processed, the approximate result is closer to the final result (which is also reflected in higher confidence and/or smaller error bounds). Online aggregation techniques use the current result (the result on the previously processed tuples) to approximate the exact result. Online aggregation is related to IVM as in both cases we maintain the result according to the tuples seen so far. This allows sharing the machinery between IVM and online aggregation. The key technical contributions of past work in the area of online aggregations are techniques for sampling from the input data set in such a way that results converge relatively quickly as well as good bounds on the current error of the approximate results can be given. Squall does not currently take advantage of sampling-based approximate query answering. However, we can easily extend Squall with machinery from the online aggregation literature [73, 107, 70, 77] for sampling the input and approximating the result.

2.2 Requirements for online systems

Low latency. In an online system, we need to preserve low latency between the time a tuple enters the system and the time the result is updated. Each tuple typically introduces a small change to the operator state and to the final result, but the tuple input rate is high. The goal is to avoid costly re-computing the result after each new tuple arrives, as it would be the case if we use an offline system. Rather, we need to preserve the state and the result from the computation performed so far. In addition, to maintain both low latency and high throughput of the system, we need to perform the computation in parallel, as well as to provide efficient local join operators. Next, we discuss parallel execution.

Parallel execution. Modern online applications typically have high input rates, and require large-state operators [37, 24] which results in high memory consumption. Thus, a single-machine setup is incapable of satisfying these requirements. Rather, we need to execute our operators on a cluster. The challenge here is to ensure correctness of the result, and to provide for load balancing even in the presence of skew. We discuss skew resilience next.

Skew resilience. Skew is ubiquitous and it appears in many real-life datasets [140, 36]. An example of skewed distribution is zipfian distribution [146], which states that a key multiplicity (the number of tuples with a particular key) and the rank of key multiplicity are inversely proportional. That is, the first key (the key with the highest multiplicity) has $n\times$ bigger multiplicity than the n-th ranked key. For instance, the first key (which is the most popular one) has $3\times$ bigger multiplicity than the third ranked key. Consequently, the most popular key corresponds to a high percentage of the entire dataset. Let us consider an operator that uses hash partitioning, as existing online systems, such as Twitter's Storm [95], Spark Streaming [143], Flink [22], most frequently use this type of partitioning. Let us assume that 20% of the entire dataset has the most popular key. Given 10 machines, ideally each machine should process 10% of the dataset. However, due to the skew, the machine which processes the most popular key is assigned $2\times$ more data (20% rather than 10%), hindering the operator performance. As we already discussed, this type of skew is called Redistribution skew (RS).

In addition to RS, join operators also suffer from Join Product Skew (JPS), which refers to uneven number of produced output tuples among the machines. Let us consider a 2-way parallel symmetric hash join [66], which is an equi-join that uses hash partitioning (it partitions the data using the hash value of the join key). Let us assume that both relations in the join have zipfian distribution with 20% of each relation corresponding to the most popular key. If this key is the same in both base relations, not only that the machine which is assigned the most popular key have to process more input tuples than the other machines, but that machine also needs to perform cartesian product among the tuples with the most popular key from the two relations. In this case, JPS affects the performance even more than RS. The challenge is to design partitioning schemes that address both RS and JPS.

Handling skew in online systems is even more important in an online system compared to an offline system. In an offline system, skew affects throughput and total execution time. In

an online system, skew degrades these performance metrics even more and it affects other metrics as well. First, overloaded nodes cannot keep up with the incoming input rate, and thus they suffer from high or even ever-increasing latency. Second, an overloaded node may run out of memory, either because network queues accepting tuples from data sources or upstream operators grow very large, or because of high number of assigned tuples. In either case, the system needs to prematurely terminate the query, or to resort to spilling to disk. As our performance numbers from Section 6.4 show, spilling to disk results in an order of magnitude performance degradation. Furthermore, spilling to disk leads to underutilization of the downstream machines. Some online systems such as Twitter Heron [81] have a back pressure mechanism, which allows each operator to limit its input rate to avoid overloading by sending control signals to the upstream operators or data sources. This mechanism bounds the size of network queues and thus avoids memory overflow on all the operators, but not on data sources. Back pressure causes underutilization of other machines (those that are assigned a smaller number of tuples) and it still results in increased latencies (the time a tuple waits in a buffer to be sent to a downstream operator also counts as latency). Overall, a single overloaded node affects both the performance and resource utilization of the whole query plan [55, 54].

Scalability. Scalability implies that, given more resources, a system or an algorithm performs more work proportionally to the given resources. In particular, weak scalability states that, when increasing the number of given machines and the dataset size proportionally (e.g., when we double both the number of machines and the dataset size), the performance (execution time) should remain approximately the same. Let us consider the hash partitioning example from above when given 2 times more machines (20 machines) and 2 times bigger dataset. Assuming that the skew degree remains the same (20% of a relation has the most popular key), after increasing the dataset size the multiplicity of the most popular key is doubled. Consequently, the machine responsible for processing this key is assigned two times more tuples than before. On the other hand, hash partitioning on a uniform dataset scales well. Thus, we need to take skew into account when designing a scalable partitioning schemes and operators. Interestingly, Stratosphere [19] project also states that their operators do not scale above certain number of machines due to data skew.

Certain system design choices can also lead to poor scalability. For instance, some existing online systems (e.g., Naiad [100]) enforce a global update order to ensure result correctness. This may not scale after certain number of machines, as there is a single entity that assigns timestamps. We discuss this line of work in further detail in Section 2.3 and compare it with Squall's design choices in Section 3.2.

Complex queries. The analytics tasks are nowadays becoming increasingly complex. That is, users want to run both 2-way and multi-way joins with complex join conditions, including both equi-join and non-equi joins [144, 45, 106, 13]. Examples include analyzing nearby objects in space or time, such as call logs analytics (e.g., base station misconfiguration) and whether forecast analytics (e.g. storm propagation). In business intelligence, we are interested

in finding customers that can afford to buy a product (e.g., the amount on their account is bigger than the product price). Unfortunately, existing work is either limited to equi-joins [13, 45], or it also supports non-equi joins but at the cost of excessive tuple replication among the machines [106, 144]. The replication leads to high amount of work performed on each machine, degrading the operator performance. The challenge is to achieve load balancing for complex operators while minimizing total work performed by the parallel operator. As we discuss later, we do so both for 2-way and multi-way joins by using the information about the join conditions and skew degree.

Adaptivity. As in an online system data statistics can change at run-time, we need to adjust the partitioning scheme accordingly (also at run-time). Unfortunately, existing work on adaptive operators supports only equi-joins and requires stalling the input streams while performing state migrations [120, 90, 91]. Rather, to keep the latencies low, we need to continue processing new incoming tuples when migrating state. Furthermore, existing partitioning schemes for complex joins (including non-equi joins and multi-way joins) [106, 144, 13] are designed for offline systems and they are non-adaptive. The challenge here is to design a general adaptive operator that adjusts the partitioning scheme at carefully chosen points in time. In particular, we need to find the right trade-off between performing too frequent state migration on one hand, and using a suboptimal partitioning scheme for too long on the other hand.

Summary. To best of our knowledge, there is no existing open-source distributed query engine that satisfies all of these requirements. On the contrary, Squall is designed to meet all the requirements, and to provide for scalable online query processing in a single system.

2.3 Existing work on online systems

This section provides a brief overview of the most important (and most widely used) existing online systems. For more details about different online systems, we refer an interested reader to an excellent survey [92].

MapReduce systems cannot provide online processing. The challenge of real-time data processing has recently moved to the forefront of interest among users of analytics and data warehousing systems as well as the large-scale Web applications / NoSQL crowd. On the other hand, map-reduce style batch processing systems [48, 7, 75] are not amenable for low-latency processing due to the following. A MapReduce job consists of a map and a reduce stage. A job does not produce any output before all the input is processed, that is, a reduce function is invoked only after the map function processes all the input data. If the computation consists of multiple MapReduce jobs, only one job is executing at a time, and the next stage blocks until the current one completely delivers its intermediate result [ii]. Thus, latencies are very high in these systems, and we need to use different systems to achieve low latencies.

Cohabitation of offline and online systems. Large Web applications companies, which play a

[ii] If there is no data dependencies among jobs, they can execute in parallel.

key role in the NoSQL movement and the development of map-reduce style batch processing systems [48, 7, 75], use batch processing systems in conjunction with large-scale realtime frontend systems. An architecture that concurrently runs fault-tolerant batch processing and low-latency online processing for the same application is denoted in literature as Lambda architecture [96]. In this architecture, once the exact results from the batch processing are in place, they overwrite the corresponding eventually consistent results from the online processing pipeline. Twitter's Summingbird [33] offers a user the same declarative interface for offline and online processing. The systems uses Scalding (Cascading's Scala API) [6] as the backend for offline processing, and Storm [95] as the backend for online processing. Summingbird also allows running the same application in both backends at the same time (hybrid mode). Google DataFlow [18] provides similar functionalities using FlumeJava framework [39] and MapReduce for offline processing, and MillWheel [17] for online processing. Google DataFlow focuses on time series data processing for unbounded streams, allowing a user to choose a tradeoff between latency, correctness and resource costs.

Micro-batch systems. There have been proposals that attempted to introduce *onlineness* in Hadoop, the most famous examples being the Hadoop Online Prototype (HOP) [46] and Scalla [88, 87]. We note that the paper [109] on Nova also claims batched incremental processing of workflows on Hadoop, but provides little detail on the systems aspects of it. HOP pipelines in small batches the map output to reducers, and it performs multi-pass merge on the reducers. However, it was shown in Scalla [88] that HOP is not amenable for high-performance online processing, because sort-merge, inherited from Hadoop, has unacceptable blocking cost. Rather, Scalla [88, 87] uses hash partitioning, which performs better. This system also maximizes performance by carefully partitioning tuples among memory and disk in the case of memory overflow. Both HOP [46] and Scalla [88, 87] focus on general micro-batch MapReduce processing, rather than on database operators. In contrast, Squall focuses on database operators. It uses hash partitioning in the case of skew-free datasets, but we design and implement other partitioning schemes as well (depending on the join conditions and skew degree).

There are attempts to bring online processing to other batch engines. Spark is an in-memory MapReduce system where the computation is specified as transformations over resilient distributed datasets (RDDs). RDD abstraction ensures that, in the case of a machine failure, other machines divide among themselves the work that was assigned to the failed machine. Spark Streaming [143, 142] is based on Spark and it simulates online processing by performing MapReduce-style computation in small batches (micro-batching). As explained in Trill [41], Spark Streaming unfortunately uses the same batch size for physical batching (which helps in achieving high performance) and semantic batching (which is due to specific window semantics). In contrast, these two types of batching are independent in Squall, and query results do not depend on the physical batch sizes. In contrast to Squall, Spark Streaming has no skew-resilient joins [iii] nor multi-way joins.

[iii] There is a spark-skewjoin library (https://github.com/tresata/spark-skewjoin) that extends Spark with the support for skew resilience for equi-joins. However, their scheme is very similar to F-Skew join [36], which handles only certain types of skew. For more information about this scheme, please refer to Section 4.5.

All these systems [46, 88, 87, 143, 142] modify an existing batch system to perform micro-batching. Micro-batch systems achieve better latencies than batch systems. However, micro-batching systems still suffer from high synchronization penalties between machines. This is due to the fact that the system needs to synchronize after each micro-batch, and new incoming tuples are blocked until the whole micro-batch is processed. If a computation contains multiple stages (MapReduce jobs), the synchronization overheads grow as the system synchronizes after each micro-batch on each stage. This is equivalent to a coarse-grained lock-step. Thus, the slowest machine of an operator limits the entire operator execution. The performance degradations occur even in the absence of skew, as one machine may be slower due to non-deterministic reasons (small glitches in network, or small differences in performance among the same hardware). Synchronization raises latencies to the order of seconds and fundamentally hinders scalability.

Ground-up online systems. Next, we describe systems that are designed specifically for online processing. These systems are implemented from scratch, rather than by modifying an existing offline system. Ground-up online systems represent the computation as a DAG of pipelined operators (rather than a series of map and reduce stages), where each operator produces output on a per-tuple basis.

Flink is an Apache project that emerged from a research project called Stratosphere [19]. This system is designed for online processing, that is, the input is unbounded stream, and the input tuples are continuously pipelined through a computation graph. However, Flink can also support offline processing by treating its input in a special way (bounded streams). Flink provides functional interface, where computation is specified through operations over parallel collections. This system offers two join partitioning schemes (repartition and broadcast) and local join operators (hybrid-hash and sort-merge). Flink is equipped with a cost-based optimizer that chooses an optimal scheme and local operator, according to the data and memory sizes. However, Flink currently does not provide skew-resilient nor multi-way joins. On the other hand, Flink has better support for UDF operators (including UDF joins) compared to Squall. In particular, Flink may reorder UDFs (and operators in general) to achieve better performance, while preserving the original program semantics. Furthermore, in contrast to Squall, Flink can run iterative analytics.

Naiad [100] provides online processing for cyclic and iterative analytics using global timestamps. A timestamp consists of location in the graph, epoch and loop counter. Naiad processes updates with tens of milliseconds latencies and it allows a user to send a tuple to a precise timestamp in the future. This system provides a high-level language support (via language-embedded query technology for .NET called LINQ) and it allows coexistance of synchronous and asynchronous computation in the same program. In asynchronous mode, tuples are sent immediately. Whereas, synchronous mode invokes a method when all the tuples for a given epoch are received (e.g., this is useful when performing an aggregation operation). Although a global notion of timestamp allows a user to express some interesting communication patterns, it limits throughput and scalability as all the tuples need to be timestamped on a single entity

in the system. We discuss this aspect of Naiad in a greater detail in Section 3.2. Finally, Naiad does not focus on supporting complex joins nor on skew resilience.

MillWheel [17] is another system for online processing. It focuses on efficient fault-tolerance techniques such as replay with duplicate elimination using Bloom filters. This work is orthogonal to Squall, as we could use MillWheel's techniques for achieving fault-tolerance techniques in Squall.

Twitter Storm [95] has a very convenient, dataflow-like, programming abstraction and excellent scalability. It allows users to write arbitrary programs by specifying the computation DAG and the code within each DAG node. Storm offers persistent storage and it supports at-least once, at-most once and exactly-once semantics. To provide exactly-once semantics, Storm uses a persistent storage. Storm's Trident library offers database operators such as aggregations, joins, selections and projections. However, Storm supports only equi-joins on skew-free datasets, as well as multi-way joins having the same join key among all the involved relations. In contrast, Squall supports complex join operators, including 2-way and multi-way joins, both over skew-free and skewed datasets. Furthermore, Storm requires a user to specify a query plan. In contrast, our system provides SQL interface, and automatically translates SQL to query plans, which are then translated to Storm topologies. Squall is based on Storm, and we discuss both systems in greater detail in Section 3.

Heron [81] is a next-generation online processing engine developed at Twitter. Heron and Storm are built with the same goal in mind, and Heron is API-compatible with Storm. In fact, Heron is built from scratch with the goal of addressing various performance bottlenecks in Storm. The main performance inefficiency in Storm is the presence of multiple levels of indirection: a worker (JVM process) has multiple executors (threads), and each executor is assigned multiple component tasks [81]. This design causes Storm to spend significant amount of time in multiplexing/demultiplexing each tuple through tasks, executors and workers. That is, each received or sent tuple in Storm goes through multiple queues and threads. In particular, a Storm worker has one thread for receiving tuples and one for sending them further down. Whereas, a Storm executor has a thread for user logic, and a thread for sending tuples to the worker. Thus, each input tuple has to go through 4 threads [81]. In addition, multiple levels of indirection result in conflicting scheduling goals and thus, in scheduling inefficiencies. By adopting a simpler design and by implementing tuple transferring more efficiently, Heron achieves an order-of-magnitude performance improvements[iv] compared to Storm. Heron also achieves better scalability than Storm due to limiting the maximum number of connections for heartbeats (Zookeeper) and for tuple routing (Stream Manager) via hierarchical structuring of communicating nodes.

Trill [41] is a high-performance library for online processing, and Quill [40] is a parallel version of Trill. Trill/Quill achieve very high throughput mainly due to using optimizations related to column stores. This line of work is orthogonal to Squall, as we could employ their optimizations

[iv] http://www.infoq.com/news/2015/06/twitter-storm-heron

in our system.

Overall, existing open-source online systems focus on distribution primitives (e.g., communication patterns, fault tolerance) and low-level performance optimizations. In contrast to existing online systems, Squall focuses on supporting complex joins and on skew resilience.

3 System architecture

3.1 Overview

Squall is an online distributed query engine which achieves low latency and high throughput. It supports full-history (incremental view maintenance) and window (stream) semantics. Squall uses Storm [95] as a distribution and parallelization platform.

The overall system architecture is shown in Figure 3.1. Next, we give an overview of various Squall concepts.

User interface. Squall offers multiple interfaces: declarative (SQL), functional (a modern Scala collections API), interactive (Scala) and imperative (Java). Similarly to Hive [125] which provides an SQL interface on top of Hadoop [125] for offline processing, Squall's declarative interface offers running SQL over Storm for online processing [i]. We support SQL because it is becoming very popular in NoSQL systems (such as Hadoop). For example, according to [118], Hive (and its SQL interface) is used for auto-generating 95% of Hadoop jobs at Facebook (only the remaining 5% is written by hand). Squall's functional interface provides for compositions of data transformations over streams. Squall also provides interactive interface built on top of the Scala REPL (Read-Eval-Print Loop) that allows a user to interactively construct and run query plans. For each of these three interfaces, Squall translates the user input to a logical query plan (see Figure 3.1). Finally, the imperative interface gives the user full control over the physical query plan. A user can run a query plan specified by any Squall interfaces either locally or on a cluster, making it easy to learn and test Squall.

Next, we illustrate running a query using different interfaces. Our declarative interface takes SQL as the input:

```
SELECT CUSTOMER.MKTSEGMENT, COUNT(ORDERS.ORDERKEY)
FROM CUSTOMER join ORDERS on CUSTOMER.CUSTKEY=ORDERS.CUSTKEY
GROUP BY CUSTOMER.MKTSEGMENT
```

[i] The name of our system contains letters S, Q and L. In addition, the names of both Squall and Storm are related to weather conditions.

A functional (Scala) interface leverages the brevity, productivity, convenience, and syntactic sugar of functional programming. For example, the previous query is represented as follows:

```
val customers = Source[customer]("customer").map { t => Tuple2(t._1, t._7) }
val orders = Source[orders]("orders").map { t => t._2 }
val join = customers.join(orders)(k1=> k1._1)(k2 => k2) //k1._1 = k2
val agg = join.groupByKey(x => 1, k => k._1._2) //count and groupby
agg.execute(conf)
```

The same query is expressed in the Squall's imperative interface as follows:

```
Component customer = new DataSourceComponent("customer", conf)
                            .add(new ProjectOperator(0, 6));
Component orders = new DataSourceComponent("orders", conf)
                            .add(new ProjectOperator(1));
                      // join on CUSTKEY
Component custOrders = new EquiJoinComponent(customer, 0, orders, 0)
              // group by MKTSEGMENT and count
              .add(new AggregateCountOperator(conf).setGroupByColumns(1));
```

Logical and Physical query plans. A logical Squall query plan is a DAG of relational algebra operators. A physical Squall query plan consists of a DAG of physical operators and their requested level of parallelism. An operator runs in parallel on multiple machines and it is specified by the partitioning scheme and local algorithm. To minimize the number of network hops, and thus maximize the performance, we co-locate the connected operators that use the same partitioning scheme. We denote a pipeline of co-located operators as a *component*. Component is a logical unit of execution in a distributed environment, and it can be scaled out to many machines. Figure 3.1 shows components as rounded rectangles in the example physical plan. An example of a component is a join followed by a selection. Both in logical and physical plan, data sources R, S and T are continuously sending tuples, and the final component is continuously performing aggregations on its incoming tuples.

Operators. Squall offers database operators such as selections, projections, joins and aggregations (we currently support SUM, COUNT and AVERAGE aggregates). In this thesis, we focus on joins, as they are the most challenging operators. A join operator consists of a partitioning scheme and a local join algorithm. We build novel partitioning schemes in Squall:

- Equi-weight histogram (EWH) scheme is presented in Section 4,
- Hypercube schemes for multi-way joins (a multi-way join runs within a single component, rather than using a pipeline of 2-way joins) is described in Section 5 and
- Adaptive 1-Bucket scheme is introduced in Section 6.2.

Squall provides novel join operators by combining each of these partitioning schemes with state-of-the-art local join algorithms. Our local join operators employ indexes that we build on

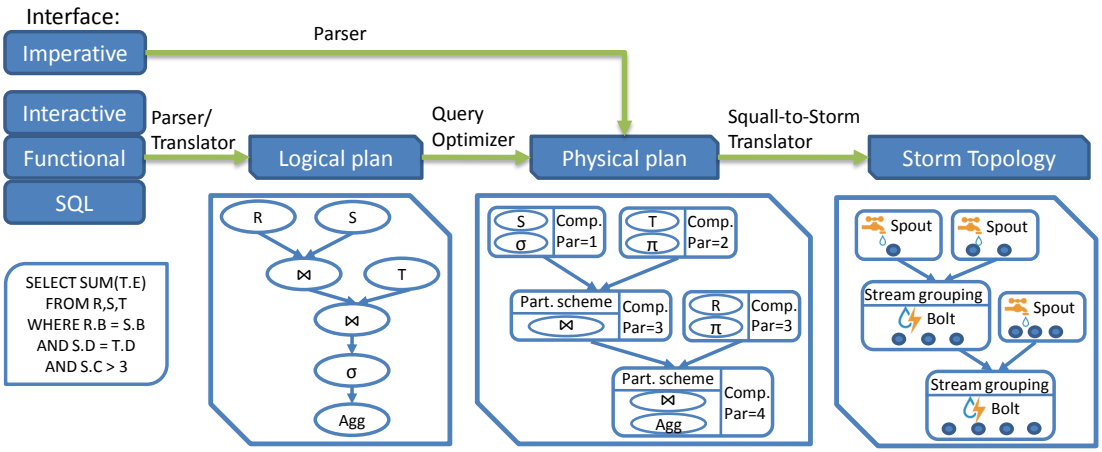


Figure 3.1: Squall architecture. An example query plan has selections (σ), projections (π), joins ($\bowtie$) and aggregations (Agg).

the fly (hash indexes for equi-joins, and balanced binary tree indexes for band and inequality joins). For example, let us consider a join condition $R.A = S.A \ \ AND \ \ 2 \cdot R.B < S.C$. In this case, Squall builds hash-indexes $R.A$ and $S.A$ and balanced binary tree indexes $R.B$ and $S.C$. Upon tuple arrival, we store the tuple, update all of its indexes, and lookup indexes on the opposite relation in order to produce result tuples.

Regarding the system optimizations for local joins, we employ collections of primitive rather than wrapper types using the Trove library [9] (we implement Squall in Java). Similarly, we store complex data types (such as Strings) as byte arrays. Both of these optimizations bring significant savings in memory consumption. As explained in [103], memory savings can translate to performance improvements. Finally, we describe local multi-way joins in Section 5.1.4.

For aggregations, there are several ways to assign them to components in a query plan. First, if a user is responsible for consuming the last component output in parallel, the last aggregation can be collocated with the previous operator (e.g., join). Second, a user may require that each machine is responsible for a single GROUP BY key of the aggregation. In that case, we need an additional component for the aggregation, unless the previous operator uses hash partitioning, and the GROUP BY key and the key from the previous operator are in Superkey/key relationship [27]. This property implies that, given that the key from the previous operator is assigned to exactly one machine (this holds due to hash partitioning), each GROUP BY key is assigned to exactly one machine. Consequently, in that case, we do not need an additional component for the last aggregation. Third, a user might want to merge all the results on a single machine. The query plan needs to scale well with the increase in parallelism of the next-to-last component. To that end, we use two aggregation operators. The first one pre-aggregates the output of the last non-aggregation operator, and these two operators are co-located on the same component. The second aggregation continuously merges all the

results, and it executes on a component with parallelism of 1. The goal of the pre-aggregation is to coarsen the input to the next (single-machine) component, so that the next component can sustain its input throughput. In particular, the first aggregation periodically (in tens or hundreds of milliseconds) sends its entire state (after which it flushes it) to next component (which has parallelism of 1). The assumption is that there is certain degree of repetition in the GROUP BY keys. Otherwise, pre-aggregation brings no performance benefits. By doing aggregation on two components, we reduce network traffic (potentially increasing throughput) and preserve scalability, but latency grows due to periodical sending from the first aggregation.

Squall provides both full-history and window semantics for its operators. It implements typical stream primitives, such as tumbling and sliding windows, by adding the window expiration logic on top of the full-history engine.

Query optimizer. Squall's query optimizer generates a physical plan from the logical plan. Squall provide a cost-based and a rule-based optimizer, as well as versions of the optimizer in which a user can manually specify parallelism and/or join ordering. In what follows, we present the cost-based optimizer. The optimizer maximizes throughput and minimizes both latency and the number of machines used. It chooses an optimal join order and an optimal parallelism for each component. An optimal component parallelism implies that machines running the component tasks are neither overloaded nor mostly idle. We refer to this as universal producer-consumer balance. The universal producer-consumer is balance is important as an overloaded machine suffers from ever-increasing latency and low throughput due to the fact that most of the time a machine is doing I/O rather than useful computation. On the other hand, keeping machines mostly idle wastes resources, which is particularly important in cloud environments that employ pay-as-you-go policies.

The optimizer automatically assigns operators to components and specifies connections between components. In particular, it starts from the data sources and adds the operators one after another, automatically pushing selections and projections as close as possible to the data sources. Where possible, the optimizer co-locates operators to components to minimize network transfers. It also performs common subexpression elimination. That is, if only expressions are used downstream a component in the query plan, the component sends only these expressions (rather than the all the corresponding fields). To do so, each component decides on its output scheme based on the fields/expressions that are needed downstream in the query plan.

The optimizer uses a dynamic programming (Selinger-style [119]) algorithm to evaluate different join orders and to pick the best plan for each intermediate result. The best plan is the one which uses a minimal number of machines. The algorithm builds a query plan bottom-up starting from data sources (whose parallelism is given), and specifies the right parallelism for each component using the producer-consumer balance (no machine is overloaded nor underloaded) and the previously chosen parallelism for the upstream components. For the subplans producing the same intermediate result, the algorithm prunes all the subplans ex-

cept the one requiring the smallest number of nodes. By doing so, the optimizer chooses an optimal join order and optimal component parallelisms *at the same time* (deciding on the component parallelisms after fixing the join order may lead to suboptimal query plans).

To compare the cost of different query plans, the cost-based optimizer needs some basic data statistics about the data being processed. Actually, a relatively small amount of information, such as relation size, number of different values in the primary key attributes, minimum and maximum attribute values and relationships among relations suffices. Squall builds statistics about intermediate relations (intermediate join outputs) in a similar way as [119]. We assume that selections applied before joins are uncorrelated, so that they do not change the computed relationships among joined relations. The required statistics might be available from the previous runs, or it is known from the application domain (e.g., each customer buys in average 10 items). If no statistics is available, a user can resort to an optimizer which allows her to specify the desired join order and component parallelisms. Finally, we consider the optimizer just an aspect of the system. Namely, we also provide operators that can adjust to changing data statistics (Section 6), and operators that are resilient to changes in certain types of data statistics such as changes in join selectivity (Section 5).

Squall supports a wide range of SQL queries, and it also supports some features which are outside the SQL Standard:

- GROUP BY (for example in an aggregation) is possible not only on a column, but also on a Projection over a tuple,
- DISTINCT can operate on multiple fields.

Currently, the SQL interface recognizes ANSI SQL syntax, and it instantiates equi-joins with hash partitioning and full-history semantics.

Online processing aspects. An online system must adapt to changing data statistics. Squall collects statistics and adjusts the operator's partitioning scheme at run-time (see Section 6.2). Furthermore, it offers multiple partitioning schemes that achieve different levels of adaptivity for different skew types (e.g., data, temporal and join selectivity skew) and degrees of skew fluctuations.

Distribution platform. Squall uses Twitter's Storm [95] as a distribution platform, but our contributions and ideas are more widely applicable. That is, all the proposed ideas are orthogonal to the underlying system (Storm), and are applicable to other systems as well (Flink, Spark Streaming etc.). In other words, although Squall uses Storm, we could have also used Spark Streaming. For our purpose, the two systems are interchangeable, even though they come from different backgrounds, Storm having been developed for realtime processing using certain stream processing abstractions, and Spark Streaming having been developed by modifying Spark, very pronouncedly a batch processing system, to perform online processing. We note that Storm is sometimes called a data stream processor, but we think of it more as

an online/realtime analytics system with a very convenient programming abstraction and excellent scalability, since it does not of itself enforce small state or handle overload situations (by load shedding). Along these lines, Akidau *et al.* [18] explain that micro-batch or streaming systems should be equivalent from the user perspective, that is, a user can use either kind of systems to run the same application. The authors state that micro-batch and streaming systems should differ only in the achieved tradeoff between latency, throughput and resource utilization.

In Storm, real-time computation is performed through topologies. A topology is a graph of spouts (data sources) and bolts (which perform computation). A spout generates a stream(s), where a stream is a sequence of tuples. A spout can read data from external entities, such as Kafka queues. A bolt performs computation; it consumes some streams and produces new ones. Each topology graph node (spout or bolt) executes on one or more machines. An edge in the topology graph is called stream grouping, and it represents partitioning of incoming tuples from a stream among the machines of a bolt. The mapping between streams and bolts is many-to-many, that is, a bolt can subscribe to multiple streams, and multiple bolts can subscribe to the same stream. Squall maps a physical query plan to a Storm topology, components to Storm spouts and bolts, and builds partitioning schemes using Storm's stream grouping. Storm takes care about assigning spouts and bolts to particular nodes in the cluster, and for communication among them.

Storm and Squall are written in JVM languages (Clojure, Java, Scala). Storm topologies are analogous to MapReduce/Hadoop jobs, but with important differences. First, a Hadoop job consists only of a map and (optionally) a reduce stage. In contrast, a Storm topology is a graph with arbitrarily many nodes (spouts and bolts), which can be connected in many different ways (stream groupings). Second, Hadoop is meant for batch (offline) processing. A Hadoop job does not produce any output before all the input is processed, that is, a reduce function is invoked only after the map function processes all the input data. A Hadoop application may contain multiple jobs, but jobs with data dependencies are executed in a serial order (a job can start only after it predecessor finishes). In contrast, Storm supports online processing, which means that the output is continuously produced as new tuples are coming to the system. To that end, all the Storm nodes run in parallel (thus competing for resources) and the output of each component is continuously sent to the input of its downstream components. As the system throughput is determined by the slowest component, it is important to assign the right spout/bolt (Squall component) parallelism.

Next, we discuss employed parallelization architecture. Shared-memory model assumes a single server machine, where multiple tasks have access to the same memory. Shared-nothing architecture implies that each task has a separate address space, and that all the communication among tasks is explicit via communication primitives. Shared-memory architecture provides for cheap communication between tasks, but it requires locking to avoid inconsistencies in the result. Furthermore, parallelism in this architecture is limited by the number of hardware threads in the server machine. In certain cases (e.g., small number of

tasks, low contention), shared-memory achieves better performance than shared-nothing architecture. However, we employ the shared-nothing architecture, due to the following reasons. First, this architecture is more general, as it can scale out beyond one server machine. Second, shared-nothing architecture lends itself to scalable execution, as it avoids locking completely.

Squall is a main-memory system. It also offers connectivity to BerkeleyDB [108], which spills tuples to disk when main memory is insufficient. However, throughput and latency are orders of magnitude better when only main memory is used.

3.2 Consistency

Squall is built around highly scalable techniques for processing queries *online*. As we already discussed, online processing implies that results are computed by incrementally building the result as the input arrives. Hence, each input both produces output and updates system state necessary for processing subsequent inputs. This sort of dependent system state change is in direct conflict with our goal of distribution, which is necessary for analytics processing at large scale. These state changes follows complex patterns defined, not only by the query, but also by the state of other machines. Due to the fact that not all tuples are sent to all nodes, a node cannot distinguish a missing from delayed tuple, resulting in a lack of liveness property. This problem is a variant of the FLP impossibility result [59]. In such environments, data might become inconsistent, thus generating incorrect query results. (The last part of Section 6.2.3 shows examples of data arrival patterns which lead to incorrect results if no consistency-enforcing mechanism is used.) To preserve correctness, existing online systems (e.g., Naiad [100]) employ a suitable consistency-enforcing mechanism, typically through enforcing a global update order. It has been frequently observed that classical strong consistency protocols for keeping partitioned state in sync do not scale to large distributed systems (e.g. [34, 133]). Recently, some very large-scale systems have been built as hybrid systems that combine expensive and lightweight consistency protocols. They employ strong consistency protocols, but only to address technical subproblems that do not need large-scale distribution, splitting the scaling challenge into a small subsystem that uses strong consistency protocols and a separate large subsystem that handles the bulk of the work and does not require expensive protocols. An example of this approach is the Google File System (GFS) [62].

Squall aims at distributed low-latency online query evaluation; the goal is to develop a scalable eventual consistency [133] mechanism that does not make heavy use of strong consistency protocols. The principal challenge is to perform distributed binary operations, specifically joins in a non-batched, online, and asynchronous fashion that does not require strong consistency methods to keep nodes in lockstep, while achieving high scalability with low update (view refresh) latencies.

Thus, the key technical problem is the distributed online evaluation of joins. Squall circumvent the problem of inconsistencies for both 2-way and multi-way joins by carefully partitioning

the state, rather than by preserving a global order. Update processing is parallelized across partitions such that eventual consistency of the result is guaranteed: Inside a partition, updates are implicitly ordered, whereas updates from different partitions cannot affect each other (they produce independent output tuples). In other words, we ensure that all the join results can be generated by joining tuples stored locally on each join machine, so there is no danger of violating atomicity. By composing these joins into a query plan (pipeline of joins), we preserve eventual consistency on the level of the entire query plan.

For skew-free 2-way equi-joins, all the tuples from both base relations having the same join key value are sent to the same join machine. To that end, we use data partitioning in the style of an online MapReduce reshuffler to ensure eventual consistency of query results. However, MapReduce uses sort-merge which is blocking, while Squall uses non-blocking partitioning (e.g., hashing). Tuples can be processed in any order, as the output is the same for the same set of incoming tuples (join is a commutative and distributive operation).

For 2-way equi-joins over skewed data, 2-way non-equi joins and multi-way joins, we assign tuples to join machines such that each output tuple is assigned to a single machine. This approach implies a certain degree of replication, as we will see in the following two chapters. However, we believe that replication is fundamentally more scalable than enforcing a global order, as it lends itself to making progress independently among machines running tasks in parallel. This motivates our work in the following two chapters on minimizing replication for joins.

Stable consistency. Our system guarantees eventual consistency [133]: although the same correct final result is always eventually produced, results may be temporarily incomplete, since some updates may have not yet fully propagated. However, there are cases in which a user may want strongly-consistent results periodically, or upon a request. Strong consistency implies that the final result is consistent with respect to a global snapshot comprising a subset of the input tuples arrived to the system. We achieve strong consistency with respect to all the input tuples arrived before a certain point in time. We do so in a decentralized, non-blocking fashion using punctuations. In particular, upon request a punctuation is sent all data sources. Then, each data source inserts the punctuation in its output streams, forwarding it to all its downstream component tasks. We rely on in-order communication between any pair of tasks (e.g., TCP is used). Each component task maintains separate data structures for tuples arriving before and after punctuation (old and new epoch) for each of its upstream component tasks. The stable result is consistent with all the input tuples labeled with the old epoch. The output of a component task is marked with the old epoch only if all the contributing tuples belong to the old epoch. Otherwise, an output tuple is labeled with the new epoch. After a component task receives a punctuation from each of its upstream component tasks, it sends punctuation to all of its downstream component tasks. A punctuation received from an upstream component task implies that no new tuples for the old epoch (and stable result) will come from that task. The last component also produces results labeled with the old and new epoch, and once it produces punctuation, the stable result is completed. After that, each

component task merges different states (old and new epoch for each upstream component task), and waits for new punctuation signals.

4 A partitioning scheme for 2-way Joins

We address the problem of accurate and efficient load balancing for parallel offline joins. (The discussion on how to take an offline operator and turn it into an online one is in Sections 6.2 and 6.3.) We show that the distribution of input data received and the output data produced by worker machines are both important for performance. As a result, previous work, which optimizes either for input or output, stands ineffective for load balancing. To that end, we propose a multi-stage load-balancing algorithm which considers the properties of *both* input and output data through sampling of the original join matrix. To do this efficiently, we propose a novel category of *equi-weight* histograms. To build them, we exploit state-of-the-art computational geometry algorithms for rectangle tiling. To our knowledge, we are the first to employ tiling algorithms for join load-balancing. In addition, we propose a novel, join-specialized tiling algorithm that has drastically lower time and space complexity than existing algorithms. Experiments show that our scheme outperforms state-of-the-art techniques by up to a factor of 12.

4.1 Introduction

There is an increasing demand for scalable and efficient parallel processing of large amounts of data. Load balancing is crucial for reaching this goal, as the total execution time depends on the slowest machine. In this chapter, we develop algorithms and techniques for efficient and accurate load balancing for parallel joins. We design and implement a partitioning scheme which assigns input tuples to machines such that each machine performs approximately the same amount of *minimal* join work.

Join types. The state-of-the-art parallel *equi-joins* rely on hashing with special handling for heavy hitters (join keys with high multiplicity). Examples are the PRPD [140] and F-SkewJoin [36] schemes. Beame *et al.* [29] prove that a modified PRPD scheme [140] is close to the communication optimum.

Unfortunately, these approaches are limited to equi-joins. In contrast, we propose a par-

titioning scheme for a broad class of *monotonic joins* [106] that include combinations of equality, band and inequality ($<$, $\leq$, $>$, $\geq$) join conditions (e.g., band-join is a combination of 2 inequality join conditions). Still, for joins with only equality conditions, one should use existing approaches (e.g. [29]).

Monotonic joins often arise in practice. Notable examples of band-joins are time-distance joins (e.g. in call logs [144]) and space-distance joins (e.g. in locating nearby objects [106]).

Skew types. Data skew occurs frequently in industrial applications [140, 36]. Load balancing is challenging in the presence of two major types of skew [134]. First, *redistribution skew (RS)* represents uneven input data partitioning among the machines due to skew in the join keys. Thus, RS impedes performance. For instance, the 1-Bucket scheme [106] achieves up to 5× speedup by addressing RS.

Second, *join product skew (JPS)* [134] represents imbalance in load due to variability in the join selectivity, causing disproportionate numbers of output tuples to be processed among parallel workers. That is, a few machines produce a large portion of the output. These machines become bottleneck, severely hindering performance. In fact, JPS can occur even in the absence of RS [112]. The following example illustrates that.

Example 4.1.1. *Let us consider a band join with condition* $|R_1.A - R_2.A| \leq 10$. *Let us consider the bucket with range* [0..30] *assuming that each relation has* 10 *join keys in this range. If* $R_1.A$ *and* $R_2.A$ *never satisfy the join condition (e.g.* $R1.A$ *in* [0..9] *and* $R2.A$ *in* [20..29]*), the output size is* 0. *On the other hand, if each of* $R_1.A$ *and* $R_1.A$ *has* 10 *distinct values in* [0..9]*, the output size is* 100.

Although each bucket has the same size $b_s = 10$ (there is no RS), the join output size per bucket varies from 0 to $b_s^2 = 100$ (there is JPS), depending on the the *relative* distribution of the join keys between the input relations. Thus, using only bucket sizes leads to inaccurate estimation of the output distribution, which results in JPS.

Depending on the input and output sizes, JPS may impede performance more than RS. Our evaluation shows that our scheme, which addresses both RS and JPS, achieves up to 12× speedup compared to a state-of-the-art scheme which addresses only RS [106].

Previous work. We present approaches that, similar to ours, go beyond just equi-joins and also support band-joins, inequality-joins etc. We classify previous work as follows. *JPS-avoidance* schemes (e.g. 1-Bucket [106]) balance the output-related work among the machines, regardless of the join selectivity. However, these schemes heavily replicate the input tuples, causing high network and memory consumption, high input-related work per machine, and thus, high execution time. *JPS-susceptible* schemes (e.g. M-Bucket [106]) do not estimate the join output distribution. Hence, these schemes cannot address JPS, causing high output-related work per machine and a vast disparity in the amount of work assigned to machines. In general, previous work does not capture the output distribution, as this requires the output sample. Building

the sample is hard, as a join between uniform random samples from the input relations is not a uniform random sample of the join output [43].

Our scheme. We propose a novel partitioning scheme which eliminates both RS and JPS. As Table 4.1 shows, we are the first to provide a scheme which is *both* input- and output-optimal. In contrast to previous work, our scheme achieves load balancing on *minimal* work per machine, which includes both input- and output-related work. This results in better execution times.

To build such a partitioning scheme [i] , we solve two problems. First, we propose an efficient parallel scheme for capturing the output distribution. We represent the input and output distribution as a matrix, where each dimension of the matrix corresponds to the join keys from an input relation. Second, using these distributions, we optimally assign portions of the matrix (called regions) to machines.

To do so, we introduce a novel family of histograms which we call *equi-weight histograms*, and a novel *histogram algorithm* to build them. An equi-weight histogram is a partitioning of the matrix into regions where regions have almost the same weight (the region weight corresponds to the machine's work). Thus, a partitioning scheme based on the equi-weight histogram *by design* provides for accurate load balancing.

Our histogram algorithm builds on state-of-the-art computational geometry (CG) algorithms for rectangle tiling. To our knowledge, we are the first to employ CG algorithms for join load balancing. Using existing CG algorithms require $\mathcal{O}(n^5 \log n)$ time to produce an accurate partitioning (n is the input relation size). This is impractical, as it is more costly than executing the join itself. In contrast, our algorithm runs in $\mathcal{O}(n)$ time, *while* providing for accurate load balancing, close to that of the baseline CG. We achieve efficiency and accuracy as follows.

First, we devise a novel CG algorithm that employs the domain-specific knowledge about monotonic joins (the properties of the join output distribution). This algorithm drastically reduces the time and space complexity compared to the baseline CG, *while* providing the same accuracy.

Second, we devise a 3-stage histogram algorithm (sampling, coarsening and regionalization), where the output of each stage is the input to the next one in the chain. Each stage reduces the input size of the next one, while providing guarantees for its output. As later stages have more coarse-grained input, we employ more precise algorithms for them to preserve accuracy. As more precise (and expensive) algorithms work on smaller inputs, we preserve efficiency.

Third, we resolve the challenging problem of setting the right output size for each stage. Namely, the size must be small enough to keep the algorithm running time short. On the other hand, the size must be big enough, as insufficient output granularity (resolution) leads

[i] By a partitioning scheme we mean either the algorithm for generating the partitioning, or the partitioning itself. In this case we mean the latter. In general, the meaning should be clear from the context.

Table 4.1: Comparison with most important related work.

Partitioning Scheme	*Input-Optimal*	*Output-Optimal*
1-Bucket	✘	✔
M-Bucket	✔	✘
EWH(ours)	✔	✔

to inaccurate load balancing (one machine is assigned much more work than the others).

We explain highlights of our solution while outlining the main contributions of this work:

1. To provide for efficient and accurate load balancing, we devise a multi-stage histogram algorithm which contains a novel, join-specialized computational geometry algorithm.

2. Our scheme achieves *minimal* work per machine, without imposing any assumptions about the data distribution.

3. We experimentally validate our scheme. Compared to state-of-the-art, our scheme achieves up to 12× speedup in terms of total time (which includes both building the scheme and performing the join) and is up to 5× more efficient in terms of resource consumption.

The rest of this chapter is organized as follows. First, we introduce the background and define the problem statement; §4.3 presents the algorithm for building equi-weight histograms; §4.4 shows how to incorporate the histograms in a join operator; §4.5 discusses related work and, finally, §4.6 evaluates the performance and validates the effectiveness of equi-weight histograms in achieving load balancing.

4.2 Background & Preliminaries

This chapter is organized as follows. First, we introduce definitions used throughout this chapter. Important symbols used in this chapter are summarized in Table 4.2. Then, we describe existing partitioning schemes and highlight the benefits of our equi-weight histogram scheme on a concrete join example. Finally, we define the problem statement for building our partitioning scheme.

4.2.1 Definitions

Join Model. We model a join among relations R_1 and R_2 as a join matrix $\mathcal{M}$. For row i and column j, cell $\mathcal{M}(i,j)$ represents a potential output tuple. $\mathcal{M}(i,j)$ is 1 iff r_i and s_j tuples satisfy the join condition. As the result of any join is a subset of the Cartesian product, the join matrix can model any join condition. Figure 4.1a shows a matrix for a band-join with a join condition $|R_1.A - R_2.A| \leq 1$. We focus on the joins which are common in practice, and for

Table 4.2: Summary of the notation used in the paper.

Symbol	*Description*	*Value*
R_1, R_2	Input relations	
J	The number of machines	
n	Max. input relation size	
m	Join output size	
ρ_{oi}	Output/Input ratio	
$w(r)$	Weight of region r	
$\mathcal{M}$	Original join matrix	
$\mathcal{M}_S$	Sample matrix of size $n_s \times n_s$	$n_s = \sqrt{2nJ}$
s_i	Input sample size	$\Theta(n_s \log n)$
s_o	Output sample size	$\Theta(n_s)$
$\mathcal{M}_C$	Coarsened matrix of size $n_c \times n_c$	$n_c = \Theta(J)$
$\mathcal{M}_\mathcal{H}$	Equi-weight histogram	

which state-of-the-art techniques perform poorly, that is, *low-selectivity joins*[ii]. These joins have sparse join matrices, that is, only a small portion of the Cartesian space produces output tuples.

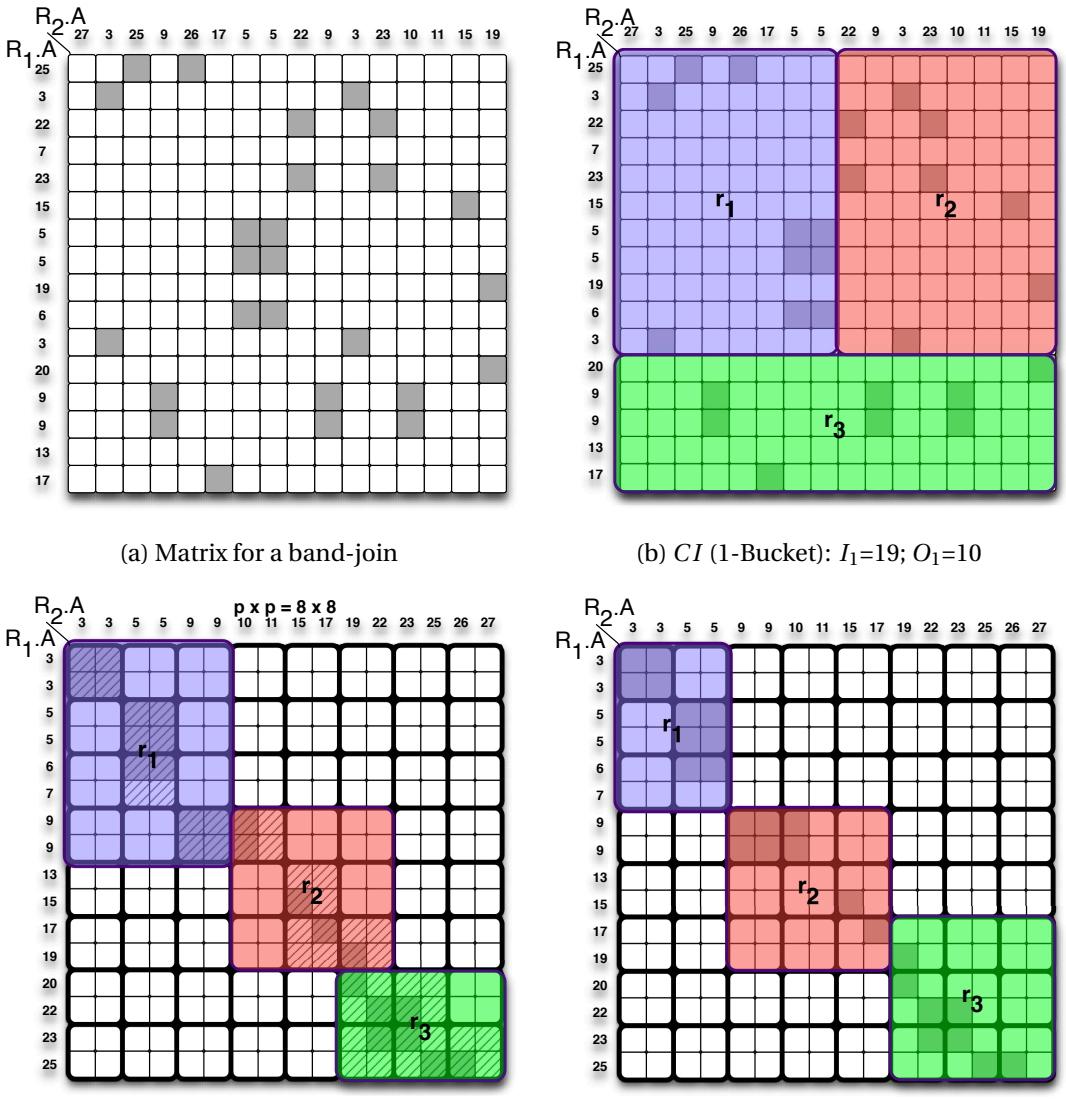
Regions. We execute a join using J machines in a shared-nothing architecture. We refer to a set of cells (that is, the corresponding input tuples) assigned to a single machine for local processing as a *region*. We adhere to rectangular regions, as opposed to rectilinear or non-contiguous regions, to incur minimal storage and communication costs [58].

***Input* and *Output* metrics.** A region's $input$ is its semi-perimeter, that is, the sum of the number of rows and columns from the join matrix intersecting the region. Processing an input tuple consists of receiving the tuple (which incurs network and demarshalling costs) and join computation[iii]. The $output$ is the number of output tuples (frequency) of a region. We use frequency as opposed to area as we focus on *low-selectivity* joins. The processing cost of an output tuple mainly comes from post-processing (writing the output to disk or transferring it over the network to the next operator in the query plan). For example, region r_1 in Figure 4.1b has $input = 19$ and $output = 10$.

Load balancing is defined as minimizing the maximum work per machine. As each machine is assigned a region, we represent the machine's work as a weight function of $input$ and $output$ costs: $w(r) = c_i(r) + c_o(r)$. As these costs depend on the local join algorithm and hardware/software architecture, $c_i(r)$ and $c_o(r)$ naturally mimic the actual cost of processing $input(r)$ and $output(r)$ tuples, respectively. Thus, the load-balancing goal can be expressed as minimizing the maximum $w(r)$. Next, we discuss whether different schemes achieve this goal.

[ii] We also run high-selectivity joins with minimal overhead (see Section 4.6.5).

[iii] The computation cost can partially belong to $output$ (see Section 4.6.1).



(a) Matrix for a band-join

(b) CI (1-Bucket): I_1=19; O_1=10

(c) CS_I (M-Bucket): I_1=14; O_1=14

(d) CS_{IO} (EWH): I_1=10; O_1=10

Figure 4.1: Different partitioning schemes (of 3 machines) on a band-join with a join condition $|R_1.A - R_2.A| \leq 1$. Shaded cells represent output tuples. (b)-(d) I_r is $input$ and O_r is $output$ metric of a region r with maximum weight $w_{x \in 1..J} = I_x + O_x$.

4.2.2 Content-Insensitive Partitioning Scheme

The *content-insensitive* partitioning scheme, CI (called 1-Bucket in [106, 58]), illustrated in Figure 4.1b, assigns all cells (n^2 of them) to machines, regardless of the join condition. Thus, regions cover the entire join matrix. This ensures result completeness and avoids expensive post processing or duplicate elimination. An incoming tuple from R_1 (R_2) randomly picks a row (column) in the join matrix, and is assigned to all the regions which intersect with the row (column). The choice of a row or a column is completely random, and it does not depend on

the tuple at all (this is why the scheme is called content-insensitive). For example, the tuple with join key 17 from R_2 randomly picks column 5, which intersects with regions r_1 and r_3. Thus, the tuple is assigned to these regions.

The scheme achieves almost perfect load balancing for $output$ by ensuring that regions have almost the same area. In particular, due to random tuple distribution, almost equal-area regions have almost equal $output$.

However, CI incurs prohibitively high $input$ costs for *low-selectivity* joins. Namely, as this scheme assigns all the cells (regardless of whether they produce an output tuple) to machines, CI suffers from excessive input tuple replication.

We compare partitioning schemes in Figures 4.1b, 4.1c and 4.1d using the weight function $w(r) = input(r) + output(r)$. Due to excessive tuple replication, CI (Figure 4.1b) has the highest maximum $w(r)$ ($w(r_1) = 29$ compared to $w(r_1) = 28$ and $w(r_1) = 20$ of the other schemes). In fact, CI works well *only* if the $output$ costs are much bigger than the $input$ costs, as in that case input tuple replication has small effect on the work per machine.

4.2.3 Content-Sensitive Partitioning Scheme

A *content-sensitive* scheme addresses the excessive tuple replication problem. It assigns an input tuple to a machine(s) according to its content (join key).

CS_I (Figure 4.1c), called M-Bucket in [106], is a *content-sensitive* scheme that uses the input statistics. To simplify notation, we denote both relation sizes as n [iv]. CS_I builds approximate equi-depth histograms[v] with p buckets over join keys of each input relation ($p < n$), and creates a grid of size $p \times p$ over the join matrix. In Figure 4.1c, $p = n/2 = 8$, and each grid cell contains $h = (n/p)^2 = 4$ matrix cells. We denote a grid cell which may produce an output tuple as a *candidate cell* (marked with diagonally engraved lines).

To efficiently check if a grid cell is a candidate (in $\mathcal{O}(1)$ time), CS_I requires a join condition that allows candidacy-checking by examining only join keys on the grid cell boundaries. This holds for monotonic joins, as the boundary join keys are sorted. For example, grid cell $(0,1)$ in Figure 4.1c is non-candidate, as the distance between the lower R_2 and upper R_1 cell boundary join keys ($5-3=2$) exceeds the width of the band-join (1).

CS_I optimizes the $input$ costs, as it assigns only candidate grid cells to machines, safely disregarding large contiguous portions in the join matrix that produce no output. CS_I scheme assigns tuples to regions according to intersection of the rows and columns with regions. For instance, in Fig. 4.1c, a tuple from R_1 with join key 9 is forwarded to regions r_1 and r_2. Whereas, all other tuples from R_1 are forwarded to exactly one region. However, as regions are rectangular, CS_I also assigns *some* non-candidates to machines. For example, although

[iv] Our analysis also holds when the sizes differ.

[v] For the sake of this example, we assume the exact histogram.

grid cell (0,1) in Figure 4.1c is non-candidate, it is assigned to r_1. In Figure 4.1, the maximum $input(r)$ of CS_I is only $I_1 = 14$, compared to $I_3 = 21$ of CI. The gap between the two schemes deepens with increasing the number of machines J, as the number of non-candidates grows.

However, CS_I is susceptible to JPS, as it ignores the actual number of output tuples ($output$) and assigns a constant to each candidate cell. In practice, the $output$ of a grid cell varies from 0 to the size of the Cartesian product between the encompassed input tuples from the two relations, that is, from 0 to the grid cell area h ($h = 4$ in our example). In Figure 4.1c, regions r_1 and r_2 have the same number of candidate cells (4), but vastly different number of output tuples (14 versus 5, respectively). This is why the maximum $w(r)$ in CS_I ($w(r_1) = 28$) only slightly improves that of CI ($w(r_1) = 29$). Thus, JPS prevents CS_I from performing better compared to CI. In fact, CS_I works well *only* if the $input$ costs are much bigger than the $output$ costs, as in that case JPS marginally affects the work per machine.

4.2.4 Equi-Weight Histogram Scheme

We propose a novel, equi-weight histogram scheme, CS_{IO} (Figure 4.1d), which achieves the best of both worlds: it avoids *both* JPS and excessive tuple replication.

CS_{IO} is a *content-sensitive* scheme that accurately estimates the number of output tuples per candidate cell via sampling (see Section 4.4.1)[vi]. In contrast to CS_I, CS_{IO} is resilient to JPS. This is why the maximum $w(r)$ in CS_{IO} ($w(r_1) = 20$) is much smaller than that of CS_I ($w(r_1) = 28$). In practice, the gap between the two schemes is much deeper. Namely, to have acceptable time for building the CS_I scheme, it must hold that $p \ll n$ [106]. This increases the candidate grid cell area h, making CS_I much more prone to JPS.

An optimal partitioning scheme minimizes the maximum $w(r)$. In contrast to CI and CS_I which work well only if $output$ or $input$ costs dominate, our CS_{IO} is close-to-optimum for a wide range of $output/input$ costs. We build such a scheme using a novel *equi-weight* histogram. The histogram contains buckets of almost equal weight, where each bucket corresponds to a rectangular region. Figure 4.2 depicts weight histograms for different schemes from Figure 4.1. CS_{IO} is based on equi-weight histograms and it is the only scheme that minimizes the maximum region weight (machine's work), providing *by design* for accurate load balancing. Next, we formalize the histogram construction problem.

Problem definition. *Given a sparse matrix $\mathcal{M}[1..n, 1..n]$ with cell values {0, 1}, partition it to $J \ll n$ non-overlapping axis-parallel rectangular regions $r_j \in \mathcal{R}$, such that each 1-cell is covered by exactly one region, and each 0-cell is covered by at most one region. The goal is to minimize the maximum weight of a region, that is* $\min\{\max_{r_j \in \mathcal{R}}\{w(r_j)\}\}$. *$\mathcal{M}$ is the original join matrix where* 1-cell*s represent output tuples and* 0-cell*s depict empty entries, $w(r)$ represents the work of a machine assigned to region r, and J is the number of joiners (machines).*

[vi] For the sake of this example, we assume exact statistics.

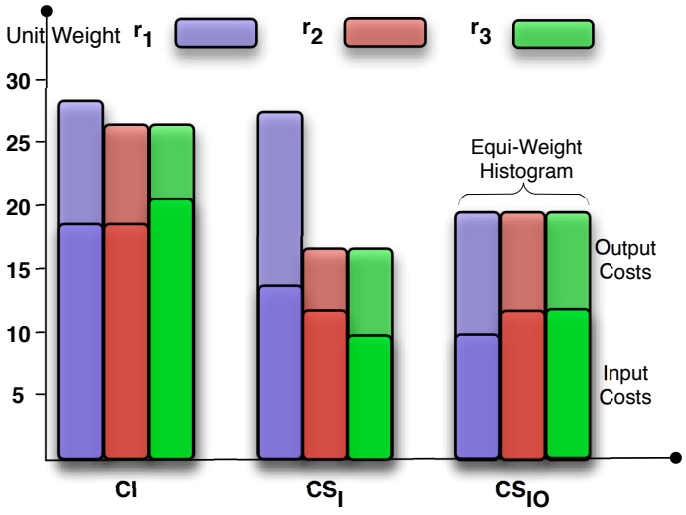


Figure 4.2: Weight Histograms.

The join matrix is just a model. The histogram algorithm does not build $\mathcal{M}$, as this is the actual join result (this would defeat the purpose of building the scheme for parallel join execution). Rather, we resort to sampling (see next section, particularly Section 4.3.1).

4.3 Histogram algorithm

In this section, we show how our efficient histogram algorithm achieves accurate load balancing. We first provide a high-level overview of the different stages of our algorithm. Then, we provide more details about each stage in subsequent sections.

Previous work. The histogram construction problem is NP-hard. The best known approximate algorithm is BSP [30], a tiling algorithm which runs in $\mathcal{O}(n^5)$ time and has an approximation ratio 2.

To create a histogram with J buckets (regions), we need to perform a binary search over the BSP (see Section 4.3.3). We denote the entire process as *regionalization*, and it takes $\mathcal{O}(n^5 \log n)$ time. This is impractical, as it is more costly than the join.

Our solution. We propose a histogram algorithm that takes $\mathcal{O}(n)$ time on a single machine, *while* providing for load balancing that is close to the one of the BSP [30]. Our idea is twofold. First, we reduce the input matrix size of the regionalization (originally, regionalization takes the original matrix $\mathcal{M}$ as the input). Second, we drastically improve the regionalization running time and space by using a novel tiling algorithm which we call MONOTONICBSP. These ideas allow us to be the first to use tiling algorithms for join load balancing.

1. Reducing the regionalization input. We introduce the *sampling stage*, which generates

Table 4.3: The time complexity improvements.

BSP over $\mathcal{M}$	*BSP over* $\mathcal{M}_S$	*BSP over* $\mathcal{M}_C$	*MonotonicBSP over* $\mathcal{M}_C$
$\mathcal{O}(n^5 \log n)$	$\mathcal{O}((nJ)^{2.5} \log n)$	$\mathcal{O}(n^{5/3} \log n)$	$\mathcal{O}(n)$

sample matrix $\mathcal{M}_S$ of size $n_s \times n_s$. $\mathcal{M}_S$ has much smaller size than the original matrix $\mathcal{M}$ ($n_s \ll n$). To provide for load balancing, n_s needs to be (at least) $\sqrt{2nJ}$ (see Section 4.3.1). If the regionalization takes $\mathcal{M}_S$ as the input, it runs in $\mathcal{O}(n_s^5 \log n) = \mathcal{O}((nJ)^{2.5} \log n)$ time. This computation cost is still too high.

To further reduce the regionalization input, we introduce the *coarsening stage*. This stage takes $\mathcal{M}_S$ as the input and creates a coarsened matrix $\mathcal{M}_C$ of size $n_c \times n_c$ ($n_c < n_s$). The coarsening reduces the regionalization input by using the distribution of $\mathcal{M}_S$ cell weights (i.e., we represent multiple small $\mathcal{M}_S$ cells as one $\mathcal{M}_C$ cell). To provide for load balancing, we opt for $n_c = 2J$ (see Section 4.3.2). If the regionalization takes $\mathcal{M}_C$ as the input, it runs in $\mathcal{O}(J^5 \log n)$ time. As using $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ machines is sufficient in practice[vii], the regionalization takes $\mathcal{O}(n^{5/3} \log n)$ time. This is still expensive compared to the join costs.

2. MONOTONICBSP: a novel tiling algorithm. In contrast to BSP which takes $\mathcal{O}(J^5 \log n)$ time, our MONOTONICBSP runs in only $\mathcal{O}(J^3 \log^2 n)$ time. To do so, MONOTONICBSP exploits the output properties of monotonic joins (see Section 4.3.3). As $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [vii], the regionalization based on MONOTONICBSP, along with the sampling and coarsening, takes only $\mathcal{O}(n)$ time (see Sections 4.3.1 to 4.3.3). Table 4.3 summarizes all the complexity improvements.

Putting everything together. Our histogram algorithm consists of 3 stages: sampling, coarsening and regionalization. Figure 4.3 illustrates the chain of the histogram algorithm stages for $w(r) = input(r) + output(r)$. The sampling stage builds $\mathcal{M}_S$ of size $n_s \times n_s$ ($n_s = 16$ in Figure 4.3a) using small input and output samples from $\mathcal{M}$. $\mathcal{M}_S$ preserves the weight distribution of $\mathcal{M}$. That is, with high probability, regions' weights in $\mathcal{M}_S$ are very close to the corresponding weights in $\mathcal{M}$. The coarsening stage creates a non-uniform grid $n_c \times n_c$ over $\mathcal{M}_S$ ($n_c = 8$ in Figure 4.3b), such that each grid cell becomes an $\mathcal{M}_C$ cell. Thus, $\mathcal{M}_C$ is of size $n_c \times n_c$. The frequency ($output$) of an $\mathcal{M}_C$ cell is the sum of the corresponding $\mathcal{M}_S$ cell frequencies, e.g. $output(\mathcal{M}_C(1,2)) = output(\mathcal{M}_S(1,2..3)) = 3$. The weight is $w(\mathcal{M}_C(1,2)) = 3n/n_s + output(\mathcal{M}_C(1,2)) = 3 \cdot 16 + 3 = 51$. The regionalization builds the equi-weight histogram $\mathcal{M}_\mathcal{H}$ by coalescing $\mathcal{M}_C$ cells into regions (see Figure 4.3c). This stage uses the hierarchical partitioning (recursively dividing rectangles into 2 sub-rectangles) over its input ($\mathcal{M}_C$). The hierarchical partitioning allows more configurations that the grid partitioning from the coarsening stage. For example, the grid partitioning cannot produce the hierarchical partitioning from Figure 4.3c, as r_1 and r_2 partially overlap over the y-axis. Section 4.7.1

[vii] Due to parallelization overhead, adding machines after a certain point provides no additional performance benefits. Our formula for J captures this observation and states that, for example, if n is hundreds of millions, it is then sufficient to use hundreds of machines.

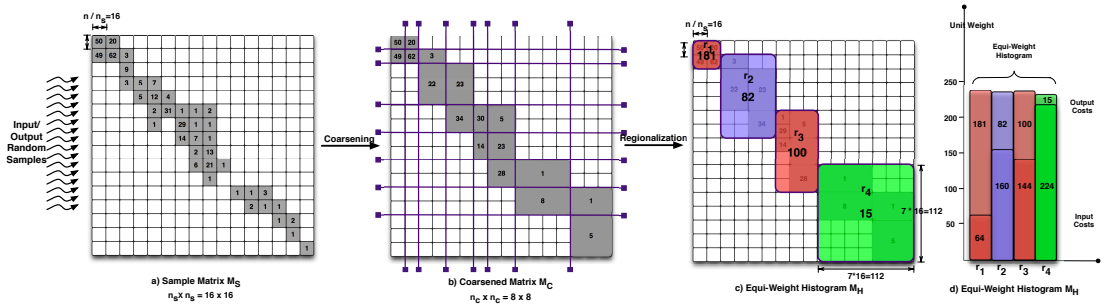


Figure 4.3: Histogram algorithm stages. The weight function is $w(r) = c_i(r) + c_o(r) = input(r) + output(r)$. For instance, in (c), $w(r_4) = 2 \cdot 112 + 15 = 239$.

illustrates different partitionings and emphasizes their differences.

The main ideas in our histogram algorithm that allow for efficient and accurate load balancing are:

1) **We avoid imposing any assumptions** about distribution within a cell, as it leads to incorrect weights and inaccurate load balancing. Thus, we create $\mathcal{M}_C$ cells (regions) on the granularity of an $\mathcal{M}_S$ ($\mathcal{M}_C$) cell.

2) **Careful choice of the matrix sizes** n_s and n_c (see Section 4.3.1-4.3.3).

3) **Using more precise algorithms when it matters** for accuracy of load balancing, that is, when the cell weights in the input matrix of the stage are high (i.e., on more coarse-grained matrices). This is important because with the increase in maximum cell weight, the potential error in load balancing is higher. In particular, as we move forward in the chain of the stages, the matrix size drops and the maximum cell weight grows. For example, in Figure 4.3, the matrix sizes are $n_s = 16$ and $n_c = 8$, and the maximum cell weights are $w(\mathcal{M}_S(1,1)) = 2 \cdot 16 + 62 = 94$ and $w(\mathcal{M}_C(6,6)) = 6 \cdot 16 + 8 = 104$. We account for this by using more precise algorithms as we move forward in the chain. Namely, the coarsening considers only grid configurations (of size $n_c \times n_c$) over its input ($\mathcal{M}_S$), as illustrated in Figure 4.3b. Whereas, the regionalization is more precise than the coarsening, as it explores all the hierarchical partitionings over its input ($\mathcal{M}_C$), as illustrated in Figure 4.3c. More precise algorithms are also more expensive per input matrix cell (see Sections 4.3.1 to 4.3.3). However, as **more expensive algorithms work on smaller input matrices** (recall $n_s = 16$, $n_c = 8$), the histogram algorithm is efficient.

4) Devising **MONOTONICBSP, a novel tiling algorithm**.

5) All the stages use a weight function which accurately estimates the processing costs, and **each stage provides guarantees for minimizing its maximum cell weight**. We prove an upper bound on the $\mathcal{M}_S$ cell weight (Lemma 4.3.1). The coarsening (regionalization) guarantees to produce partitionings within a factor of 2 from the optimum on grid (arbitrary[viii]) partitioning

[viii] It allows any partitioning into rectangles.

on the given $\mathcal{M}_S$ ($\mathcal{M}_C$) matrix. Section 4.7.1 illustrates different partitionings and emphasizes their differences.

Next, we discuss the details of each stage.

4.3.1 Sampling

This stage efficiently builds the sample matrix $\mathcal{M}_S$, which provides for accurate load balancing.

Region weight proximity. As the histogram algorithm requires precise region weights for accurate load balancing, $\mathcal{M}_S$ must preserve the region weights of the original matrix $\mathcal{M}$. As we previously saw that regions are defined by their boundary keys, a region r_s from $\mathcal{M}_S$ *corresponds to* a region r from the original matrix $\mathcal{M}$ if and only if they share all the region boundary keys. The region weight proximity means that for any two corresponding regions r_s and r, $\mathcal{M}_S$ ensures that $w(r_s) \approx w(r)$ with very high probability. In other words, any region in the sample matrix has almost the same weight as the corresponding region in the original matrix.

Previous work on sample data structures mainly concerns multi-attribute single-relation histograms, which are used for answering range queries (e.g. [99, 113]). As the algorithms for building these data structures consider *only* frequency (output), they cannot preserve the $w(r_s) \approx w(r)$ property. Hence, these algorithms fall short for providing accurate load balancing.

Building sample matrix M_S. In contrast, we build $\mathcal{M}_S$ of size $n_s \times n_s$ (in Figure 4.3a, $n_s = 16$), which keeps the $w(r_s) \approx w(r)$ property by preserving *both* the input and output distribution: *a*) To preserve the input distribution, we build an approximate equi-depth histogram [42] with n_s buckets on each input relation. The histogram boundaries form a grid of size $n_s \times n_s$ over the original matrix. Each such grid cell corresponds to an $\mathcal{M}_S$ cell. Region's $input$ is a product of the number of $\mathcal{M}_S$ cells on its semi-perimeter and the expected bucket size n/n_s. For example, in Figure 4.3a, the region defined by $\mathcal{M}_S(0..1,0)$ has $input$ of $3 \cdot n/n_s = 48$. *b*) To preserve the output distribution, we take a uniform random sample of the join output. To do so efficiently, we propose a parallel sampling scheme (Section 4.4.1). Once the sample is in place, we increment the corresponding $\mathcal{M}_S$ cell for each sample output tuple. Region's $output$ is a product of the ratio of output sample tuples within the region and the total output size m (we show how to find m in Section 4.4.1). For example, in Figure 4.3a, region $\mathcal{M}_S(0..1,0)$ has $output$ of $50 + 49 = 99$.

Efficiency and Accuracy Considerations. Setting the $\mathcal{M}_S$ size n_s is crucial for both efficiency and accuracy. As the coarsening takes $\mathcal{M}_S$ as the input, in order to keep the running time of this stage short, n_s must be small enough. On the other hand, decreasing n_s may affect accuracy of the histogram algorithm, and thus, accuracy of load balancing. In particular, if we decrease n_s, the $\mathcal{M}_S$ cells correspond to bigger portions in the original matrix and thus, the $\mathcal{M}_S$ cell weights grow. For example, the maximum cell weight in Figure 4.3a is $\sigma = 2 \cdot 16 + 62 =$

94, which corresponds to $\mathcal{M}_S(1,1)$. Assume that $\mathcal{M}_S{}'$ differs from the $\mathcal{M}_S$ in Figure 4.3a only by replacing $n_s = 16$ by $n'_s = 4$. Then, the maximum cell weight is $\sigma' = 8 \cdot 16 + 201 = 329$, which corresponds to $\mathcal{M}_S(0..3, 0..3)$ and which is much bigger than $sigma = 94$. As we avoid imposing assumptions within an $\mathcal{M}_S$ cell, regions are on the $\mathcal{M}_S$ cell granularity and a region contains at least one $\mathcal{M}_S$ cell. Thus, the maximum region weight in the partitioning scheme with $n'_s = 4$ is at least $\sigma' = 329$. Such a scheme is suboptimal compared to the $\mathcal{M}_\mathcal{H}$ scheme from Figure 4.3c,d, which has the maximum region weight $w(r_1) = 4 \cdot 16 + 181 = 245$. Thus, small n_s leads to weighty regions, which affects accuracy of load balancing.

It is challenging to choose a value for n_s, as in the sampling stage we do not know the maximum $w(r)$ of the $\mathcal{M}_\mathcal{H}$ partitioning scheme. To that end, we find a lower bound on the maximum $w(r)$ of the *optimum* $\mathcal{M}_\mathcal{H}$ scheme. We denote this lower bound as w_{OPT}, and we compute it by dividing the lower bound on the total join work ($w(\mathcal{M})$, where $input(\mathcal{M}) = 2n$ and $output(\mathcal{M}) = m$) [ix] equally among the machines. In fact, $w(\mathcal{M})$ is a lower bound as it assumes no input tuple replication. To ensure accuracy given that the coarsening and regionalization use approximate algorithms, we require that $\sigma \leq 0.5 w_{OPT}$ (rather than simply $\sigma \leq w_{OPT}$). This holds independently from the join condition and join key distribution if $n_s = \sqrt{2nJ}$, as the following lemma shows. The proofs are in Section 4.7.2.

Lemma 4.3.1. *$n_s = \sqrt{2nJ}$ is the minimum $\mathcal{M}_S$ size such that the maximum cell weight σ in $\mathcal{M}_S$ is at most half of the maximum region weight of the optimum $\mathcal{M}_\mathcal{H}$ partitioning. This holds independently from the join condition and the join key distribution, given that $m \geq n$* [x].

Next, we briefly discuss the sizes of the input and output sample, which are required for building $\mathcal{M}_S$. The detailed analysis is in Section 4.7.2. The input sample size is $s_i = \Theta(n_s \log n)$. We determine that the output sample size is $s_o = \Theta(n_s)$ using Kolmogorov's statistics [64]. Next, given $n_s = \sqrt{2nJ}$, we prove that the sampling stage has low running time.

Lemma 4.3.2. *The time complexity of the sampling stage is $\mathcal{O}(n_s \log n_s)$. For $n_s = \sqrt{2nJ}$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$* [xi]*, the time complexity is $\mathcal{O}(n/J)$.*

4.3.2 Coarsening

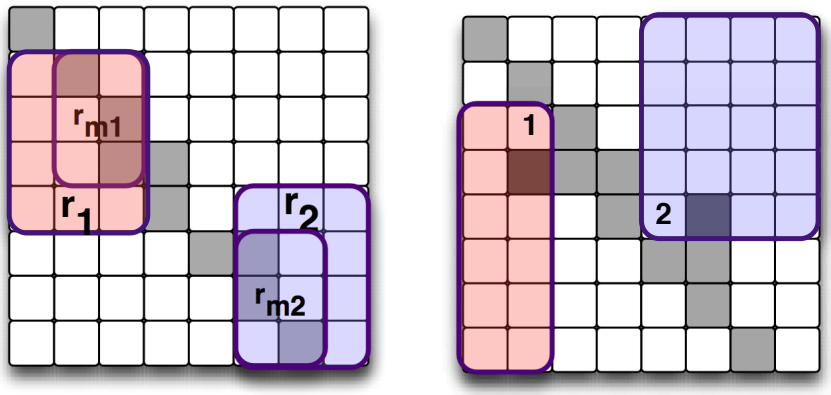
This stage creates a coarsened matrix $\mathcal{M}_\mathcal{C}$ by imposing a grid of size $n_c \times n_c$ over the input matrix $\mathcal{M}_S$. As $n_c < n_s$, the coarsening further reduces the regionalization input. Figure 4.3b shows $\mathcal{M}_\mathcal{C}$ with $n_c = 8$. The goal is to minimize the maximum cell weight in $\mathcal{M}_\mathcal{C}$. This is an NP-hard tiling problem, but it admits approximate solutions. The best approximate algorithm is the coarsening from [102], which has an approximation ratio 2.

Deciding on n_c. To keep the running time short, while achieving accurate load balancing in

[ix] It is a lower bound as it assumes no input tuple replication.

[x] This typically holds in practice. We discuss the extensions to support the general case for m in Section 4.7.2.

[xi] See footnote vii.



(a) Non-monotonic matrix (b) Min. candidate rectangles

Figure 4.4: a) Non-monotonicity in rectangles 1 and 2 due to candidates marked with black. b) r_{m1} (r_{m2}) is a minimal candidate rectangle for r_1 (r_2).

the regionalization, we opt for $n_c = 2J$. We discuss how such n_c brings accuracy in Section 4.3.4. The following lemma proves low time complexity.

Lemma 4.3.3. *The running time of the coarsening algorithm is* $\mathcal{O}((n_s + n_c^2 \log n_s) \cdot n_c \log n_s)$. *For* $n_c = 2J$ *and* $J = \mathcal{O}(\sqrt[3]{n/\log^2 n_s})$, *the time complexity becomes* $\mathcal{O}(n)$.

Monotonicity is a property of the join output distribution which holds for many interesting joins, including equi-, band- and inequality joins. It states that "if cell (i, j) is not a candidate cell, then either all cells (k, l) with $k \leq i, l \geq j$, or all cells (k, l) with $k \geq i, l \leq j$ are also not candidate cells" [106]. That is, *candidate cells are consecutive per row/column.* All the join matrices in Figure 4.1 are monotonic, while the one in Figure 4.4a is not due to the candidate cells marked black.

MonotonicCoarsening. We speed up the coarsening algorithm using monotonicity. The coarsening algorithm iteratively improves the coarsened matrix. To do so, in each iteration it computes the weights of all the $\mathcal{M}_C$ cells. As the weight of non-candidate cells is 0, it suffices to compute only the weights of the candidate cells. Monotonicity allows skipping the non-candidates for free. Thus, the MonotonicCoarsening considers only the candidate cells. This improves the algorithm's running time in practice, although the complexity does not change asymptotically.

4.3.3 Regionalization

This stage creates the equi-weight histogram $\mathcal{M}_\mathcal{H}$, which consists of (at most) J rectangular regions over $\mathcal{M}_C$ cells. Figure 4.3c illustrates $\mathcal{M}_\mathcal{H}$ with $J = 4$. The goal is to minimize the

Algorithm 1 BSP.

```
1:  function BSP(rectangles)
2:      for each rectangle r in rectangles do
3:          r_m = MINIMALCANDIDATERECTANGLE (r)
4:          if w(r_m) ≤ δ then
5:              r.regions = {r_m}
6:          else
7:              for each splitter in r_m do
8:                  {r_1, r_2} = r_m.split
9:                  splits.add({r_1, r_2})
10:             r.regions = min_{{r_1,r_2}∈splits}(r_1.regions ∪ r_2.regions)
11: end function
```

maximum region weight δ, while covering with regions all the candidate $\mathcal{M}_\mathcal{C}$ cells. [xii] The best such algorithm is *Binary Space Partition* (BSP) [30, 101]. However, BSP solves a dual problem: given the maximum region weight δ, it minimizes the number of regions. To that end, we perform a binary search over δ until BSP returns a partitioning with the available J regions (machines).

BSP [30, 101] is a tiling algorithm based on dynamic programming. It creates an optimum hierarchical partitioning, which is within a factor of 2 from an optimum arbitrary partitioning. BSP (Algorithm 1) analyzes each rectangle in $\mathcal{M}_\mathcal{C}$ as follows. If the rectangle weight is below the given maximum region weight δ, we cover the rectangle with a single region (line 5). Otherwise, BSP splits the rectangle by a horizontal or vertical line such that the total number of regions used for the two sub-rectangles is minimized (lines 7-9). We obtain a minimal set of regions for a rectangle by using the previously found splitters for each sub-rectangle. We acquire the final regions by extracting them from the rectangle encompassing the entire $\mathcal{M}_\mathcal{C}$.

Extending BSP to join load balancing. As we do not need to assign non-candidate $\mathcal{M}_\mathcal{C}$ cells to machines, we minimize a rectangle so that no candidate cell is omitted (line 3). We denote such a rectangle as *minimal candidate rectangle* r_m. For example, in Figure 4.4b, rectangle r_1 (r_2) contains no candidate cells on the left and lower (right and upper) boundaries. Thus, before computing the weights, we minimize r_1 (r_2) to its minimal candidate rectangle r_{m1} (r_{m2}).

As the input for the BSP is the coarsened matrix $\mathcal{M}_\mathcal{C}$ of size $n_c \times n_c$, and $n_c = 2J$ (see Section 4.3.2), BSP runs in $\mathcal{O}(J^5)$ time. With binary search, it takes $\mathcal{O}(J^5 \log n)$ time. As $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [xiii], the regionalization based on BSP runs in $\mathcal{O}(n^{5/3} \log n)$ time. This is expensive compared to the join costs.

BSP also suffers from high space complexity, which is proportional to the total number of rectangles in the input matrix $\mathcal{M}_\mathcal{C}$. As a rectangle is defined by 2 corners, and each corner is

[xii] As regions are rectangular, they cover some non-candidates as well, subject to minimizing δ.
[xiii] See footnote vii.

Algorithm 2 MONOTONICBSP.

```
1: function MONOTONICBSP
2:     rectangles_m = GENERATECANDIDATERECTANGLES()
3:     Sort rectangles_m according to the semi-perimeter
4:     BSPCANDIDATES(rectangles_m)
5: end function
6: function GENERATECANDIDATERECTANGLES
7:     for x_1 = 1 to n_c do
8:         for y_1 in cand. cell indexes in row x_1 do
9:             for x_2 = x_1 to n_c do
10:                for y_2 in cand. cell indexes in row x_2 do
11:                    rectangles_m.add(x_1, y_1, x_2, y_2)
12:    return rectangles_m
13: end function
14: function BSPCANDIDATES(rectangles_m)
15:    for each rectangle r_m in rectangles_m do
16:        if w(r_m) ≤ δ then
17:            r_m.regions = {r_m}
18:        else
19:            for each splitter in r_m do
20:                {r_1, r_2} = r_m.split
21:                r_m1 = MINIMALCANDIDATERECTANGLE(r_1)
22:                r_m2 = MINIMALCANDIDATERECTANGLE(r_2)
23:                splits.add({r_m1, r_m2})
24:            r_m.regions = min_splits(r_m1.regions ∪ r_m2.regions)
25: end function
```

defined by 2 $\mathcal{M}_C$ coordinates, the space complexity is $\mathcal{O}(n_c^4) = \mathcal{O}(J^4)$.

MONOTONICBSP. We propose MONOTONICBSP, a novel tiling algorithm which drastically reduces both time and space complexity of the BSP. To do so, MONOTONICBSP exploits the output distribution properties for monotonic joins. In BSP, enumerating rectangles from $\mathcal{M}_C$ results in high time/space complexity ($\mathcal{O}(n_c^4)$). However, for monotonic joins, only a small portion of these rectangles are minimal candidates. The main challenge is to enumerate only minimal candidate rectangles without even looking at all the rectangles, as this would require $\mathcal{O}(n_c^4)$ time. We do so using the following lemma.

Lemma 4.3.4. *A rectangle is defined by the upper left and the lower right corner. For monotonic joins, each defining corner of a minimal candidate rectangle is a candidate cell, yielding $\mathcal{O}(n_c^2)$ minimal candidate rectangles in total.*

As Figure 4.4) shows, rectangles r_{m1} and r_{m2} are minimal candidate rectangles, and their defining corners are candidate cells.

Thus, by designating each pair of the candidate cells as the rectangle defining corners, the MONOTONICBSP (Algorithm 2) enumerates all the minimal candidate rectangles (lines 6-13). There are $\mathcal{O}(n_c^2)$ such rectangles, as $\mathcal{M}_C$ has $\mathcal{O}(n_c)$ candidate cells (we deal with low-selectivity

joins). Then, the algorithm sorts the rectangles by their semi-perimeter (line 3), and runs a BSP version which considers only minimal candidate rectangles (lines 14-24).

Lemma 4.3.5. *The regionalization stage based on* MONOTONICBSP *runs in* $\mathcal{O}(n_c^3 \log n_c \log n)$ *time. For* $n_c = 2J$ *and* $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [xiv]*, the stage takes* $\mathcal{O}(n)$ *time.*

The space complexity of MONOTONICBSP is $\mathcal{O}(n_c^2)$, as there are n_c^2 minimal candidate rectangles (see Lemma 4.3.4).

MONOTONICBSP significantly outperforms the baseline BSP for monotonic joins, both in terms of space and time complexity. Namely, MONOTONICBSP requires only $\mathcal{O}(n_c^2)$ space and $\mathcal{O}(n_c^3 \log n_c \log n)$ time. Whereas, the baseline BSP runs in $\mathcal{O}(n_c^4)$ space and $\mathcal{O}(n_c^5 \log n)$ time.

4.3.4 Putting it all together

The computation cost. By directly applying previous work [30] (i.e., the regionalization based on BSP over the original matrix $\mathcal{M}$), computing the histogram requires $\mathcal{O}(n^5 \log n)$ time. In contrast, our 3-stage histogram algorithm runs in only $\mathcal{O}(n)$ time.

Theorem 4.3.1. *The time complexity of the histogram algorithm is* $\mathcal{O}(n)$.

The proof directly follows from Lemmas 4.3.2-4.3.5 (each stage runs in $\mathcal{O}(n)$ time).

The accuracy of load balancing. As in our algorithm the regionalization creates regions on the $\mathcal{M}_C$ cell granularity, we next discuss how much the coarsening stage affects the accuracy of load balancing. For output-only weight functions, Wang [136] shows that the arbitrary partitioning over a grid partitioning is within a factor of 4 from the arbitrary partitioning over the original data. Applied to our case, if $n_c \geq J$ and the input matrix of the coarsening stage is the original matrix $\mathcal{M}$ (rather than the sample matrix $\mathcal{M}_S$), the coarsening and regionalization produce a partitioning which is at most a factor of 4 from the one produced by the regionalization alone (this holds only for output-only weight functions). We lessen the factor of 4 by choosing $n_c = 2J$ (rather than $n_c = J$) for the $\mathcal{M}_C$ size.

Sampling minimally affects load balancing, as $\mathcal{M}_S$ with very high probability preserves the weight distribution from $\mathcal{M}$ (Section 4.3.1). Further, we minimize the effect of coarsening to accuracy by ensuring that the maximum cell weight in $\mathcal{M}_S$ is at most half of (rather than equal to) the maximum region weight in the optimum $\mathcal{M}_\mathcal{H}$ partitioning scheme (Lemma 4.3.1). We provide strong empirical evidences for the accuracy of our equi-weight histogram scheme (see Section 4.6).

[xiv] See footnote vii.

4.4 Join operator

In this section, we integrate our partitioning scheme into a join operator. First, we collect the statistics, that is, samples of the input and output tuples (Section 4.4.1). Then, using these statistics, we build the equi-weight histogram (see Section 4.3). Finally, we distribute and process the data according to the histogram.

Local Join Algorithm. Each machine processes a region using a local join algorithm. As long as all the machines run the same algorithm, our scheme is orthogonal to the local joins.

Sampling the Input Tuples. As described in Section 4.3.1, we need a uniform random sample of size s_i from each relation. We build the input sample in one pass in parallel using Bernoulli sampling [61] with a sampling rate of $q_i = s_i / n$. As explained in [61], the sample is of *expected* size s_i. In order to guarantee this sample size, we set the sampling rate q_i slightly higher. This algorithm is denoted as the *Bernoulli sampling with probabilistic sample size bounds* [61].

4.4.1 Sampling the Output Tuples

Chaudhuri *et al.* [43] show that we cannot obtain a uniform random sample of the join output by joining uniform random samples from the input relations. Alternatively, performing the entire join and then sampling from the output defeats the purpose of building the equi-weight histogram. The *Stream-Sample* algorithm [43] provides a uniform random output sample without performing the entire join. However, this is a single-machine algorithm. To make it efficient and scalable, we devise a parallel version of the Stream-Sample. Next, we discuss efficiency of this algorithm in the context of join load balancing. To our knowledge, we are the first to use random samples of the join output for parallel join load balancing. Then, we describe the baseline and parallel Stream-Sample in detail.

Efficiency. The cost of Parallel Stream-Sample, which mainly comes from scanning the input relations, is small compared to the cost of parallel join. This is due to the following:

1) The benefits of using the collected statistics easily surpass the scanning overhead, both in MapReduce [106, 56, 82] and distributed databases [111]. In both cases, scanning involves repartitioning of the join keys [106, 111]. Our experiments (Section 4.6) also show that scanning pays off, as *JPS affects performance much more than scanning*.

2) The output sample tuples contain only join keys, as we use the samples only for building $\mathcal{M}_S$ (and not for propagating it further in the query plan). This reduces the network traffic.

3) The output sample size is much smaller than the input relation size ($s_o = \Theta(n_s) = \Theta(\sqrt{nJ}) \ll n$).

Stream-Sample. The Stream-Sample [43] works only for equi-joins, but we extend it to work for band- and inequality joins as well.

First, we introduce the notation. The base relations are R_1 and R_2 and a sample from R_1 is S_1. Given a tuple $t_1 \in R_1$ with a join key $t_1.A$, the *joinable set* of t_1 consists of all the tuples from R_2 which are joinable with t_1. For equi-joins, the joinable set of t_1 comprises of all the tuples from R_2 with $t_1.A$ as the join key. For band- and inequality joins, the joinable set contains all the tuples from R_2 with a join key within a certain distance (specified by the join condition) from $t_1.A$. We denote the joinable set size as $d_2(t_1.A)$, and the ensemble of them as d_2. Using the keys from d_2 with an equality condition yields d_{2equi}. WR (WoR) is sampling With (Without) Replacement.

The Stream-Sample algorithm works as follows. We take a WR weighted sample S_1 of size s_o from R_1, where the weight of $t_1 \in R_1$ is $d_2(t_1.A)$. Then, for each $t_{s_1} \in S_1$, we randomly choose $t_2 \in R_2$ from the joinable set of t_{s_1} and produce an output tuple $t_{s_1} \bowtie t_2$.

Parallelization. We design a parallel, scalable version of Stream-Sample, which runs efficiently on the same number of machines as the join itself. For the ease of presentation, we describe it in terms of MapReduce [48] jobs.

1. We build d_{2equi} from R_2 in a single MapReduce job. To reduce the work, we designate R_2 to be smaller of both relations. To partition the work evenly, we assign the R_2 tuples to the machines according to their join keys and the approximate equi-depth histogram on R_2.

2. In this step, we build d_2 and S_1. We create d_2 as follows: Each reducer obtains a range of sorted join keys from d_{2equi} along with their multiplicities. As mentioned before, $d_2(t_1.A)$ is the sum of multiplicities of the join keys from R_2 which are within a certain distance from $t_1.A$ (according to the join condition). Each time a reducer moves in the sorted key sequence such that the joinable set changes (adding or removing a tuple), a new d_2 key-value pair is created.

We also build a WR weighted sample S_1 from R_1, where weights are based on d_2. To do so, we use a parallel one-pass algorithm for WoR weighted sampling [57] which works as follows: It puts each $t_1 \in R_1$ into a priority queue of size s_o using the priority computed as a function of $d_2(t_1.A)$. According to [57], the precise formula for priority is $r^{(}1/w)$, where $r = random(0,1)$ and w is the weight, which is in our case $d_2(t_1.A)$. After each reducer produces its Max-Heap reservoir, we merge them into a single reservoir using the same priority function. Finally, we transform S_1 from a WoR to a WR sample using [43].

We build d_2 and S_1 together in a single MapReduce job. We assign the d_{2equi} and R_1 tuples to the machines according to their join keys and the approximate equi-depth histogram on R_1 due to the following reasons: First, d_{2equi} tends to be much smaller than R_1. Secondly, by doing so, we balance the work for computing S_1.

3. Finally, we produce a uniform random output tuple for each tuple $t_{s_1} \in S_1$. As we use the output tuple only for building $\mathcal{M}_S$, it contains only a concatenation of the join keys. This relieves us from choosing uniformly at random an R_2 tuple from the joinable set of t_{s_1}, which would require processing R_2 again. Instead, we randomly choose a join key from the joinable

set of t_{s_1}, with probability directly proportional to the key multiplicity. These multiplicities are available in d_{2equi}. As S_1 is typically much smaller than d_{2equi}, we assign S_1 and d_{2equi} to the machines according to their join keys and the partitioning of d_{2equi} from step 1. Thus, we sort S_1 and use a Map-only job for this step.

Synergy. We first build equi-depth histograms on R_1 and R_2. Then, we sample input and output tuples in parallel by sharing mappers (sampling the input requires only one reducer). If an input relation has a predicate which filters out many tuples, we reduce the scanning overhead for the join by materializing the filtered relation in the statistics scan.

Parameters. To build the sample matrix, we need to know m (see Section 4.3.1). On the other hand, it is hard to obtain m ahead of time. We obtain m from the Parallel Stream-Sample algorithm. In particular, as we iterate over the entire R_1 relation in the step 2 of the algorithm, we compute m as $\sum_{t_1 \in R_1} d_2(t_1.A)$.

4.4.2 Discussion and Generalization

System architecture. As [45] shows, systems designed for main-memory parallel processing are very popular nowadays (e.g. Shark-Spark [139], Dremel [97]), mainly because of superior performance compared to the disk-based systems. For that reason, recent parallel joins are main-memory operators [111, 45]. We follow the same reasoning and implement our operator in a main-memory parallel system.

Input relations are not necessarily base relations. Rather, a join may contain selection predicates, or it may consume the output from another join. To support these general joins, we build our scheme for each join (i.e., no reusing among different joins), and report this in the total execution time. The M-Bucket scheme [106] adopts the same approach.

Multi-way joins. Our scheme assumes 2-way joins. As our scheme enhances performance especially when the $output$ cost matters a lot (e.g. transferring tuples between operators over the network), a multi-way join can be efficiently executed using a sequence of our 2-way joins. Squall also offers an operator that performs a multi-way join within a single communication step, which we present in Chapter 5.

Heterogeneous clusters. In heterogeneous clusters, we assign work to machines proportionally to their capacity. To do so, we set the number of regions (J) in the histogram algorithm higher than the number of machines.

4.5 Related Work

Load balancing is extensively studied both in MapReduce and distributed databases. There has been much work done towards devising efficient join algorithms using the MapReduce framework. The predominant join type in MapReduce is *repartition join* [32], which moves

each input tuple over the network. In distributed databases, data is already partitioned among the machines (rather than being stored externally, e.g., on HDFS, as in MapReduce). Thus, some tuples can stay on the same machine. *Broadcast join* replicates one relation on all the machines. This is efficient only if the replicated relation is very small [140]. *Directed join* moves portions of one relation to the corresponding locations of the other relation. It typically requires that one relation is physically partitioned by the join key [32]. This is a limitation when we join a relation with other relations using different join keys [32]. We propose a *novel repartition join*, as repartition join is the most widely applicable join type. Using our join as a directed join (the data is already in place) is also possible. After running our equi-weight histogram, we can assign regions to machines in such a way to maximize locality (data kept on the same machine) and minimize network traffic. This algorithm is described in the context of Adaptive Equi-weight histogram scheme in Section 6.3.2.

1. Equi-joins. Most previous work focuses on equi-joins [13, 32, 140, 36, 29, 111, 45] and partitions the input through some variant of hashing. One should use these techniques for joins that have only equality join conditions.

Next, we discuss why hashing techniques fall short for monotonic joins on an example of a band-join. Namely, hashing scatters neighboring join keys, so that the corresponding tuples from the opposite relation need to be replicated. For a band-join with the width of the band of β, each tuple from the opposite relation goes to $2\beta+1$ machines ($hash(key-\beta)$, $hash(key-\beta+1)$, ... $hash(key+\beta)$ [xv]). This implies more input-related work, as well as higher network and memory consumption. The overheads grow proportionally to the width of the band β. Range partitioning avoids this problem, as neighboring join keys are in most cases on the same machine. This leads to less tuple replication, and less overall work compared to hash partitioning.

2. Monotonic joins. In this chapter we focus on monotonic joins that do not contain only equality join conditions. State-of-the-art techniques in MapReduce are the 1-Bucket and M-Bucket schemes [106]. We discuss these techniques in detail in Sections 4.2.2 and 4.2.3. In contrast to the 1-Bucket scheme [106], our scheme achieves load balancing on *minimal* work per machine. In contrast to the M-Bucket scheme [106], we address JPS. In the context of distributed databases, Stamos *et al.* [121] present a method that covers the entire join matrix with regions, similarly to the 1-Bucket scheme. This method uses a heuristic model to minimize total communication cost. DeWitt et al. [51] study band-joins with the goal of minimizing disk accesses. Similarly to the M-Bucket [106], this work do not estimate the output distribution, and hence suffers from JPS. Finally, Zhang *et al.* [144] extend Okcan's work to evaluate multi-way joins. We dedicate Chapter 5 to multi-way joins.

3. Reliability. Bruno *et al.* [36] introduce the term reliability for equi-joins, arguing that the repartitioning overhead "is more predictable" than the imbalance in load due to JPS. We use a

[xv] This is an upper bound on the number of machines, as different hash values can be assigned to a single machine.

similar argument for monotonic joins: the sampling overhead is more predictable than the imbalance in load when JPS is not addressed. In particular, sampling introduces overhead of up to 0.13×, as Section 4.6.5 shows. However, this is acceptable compared to the speedups that our scheme achieves by addressing JPS (up to 12×, see Section 4.6.2).

Adaptive load balancing. Adaptive skew handling exist for hash joins (e.g., [71], and for general-purpose MapReduce applications (e.g., [83]). These techniques in general work as follows. When a task becomes idle, it takes over some work from the busiest task. This implies moving the tuples over the network multiple times (first to the "busy", then to the "idle" task), which increases the input-related work. In contrast, we ensure that after building the partitioning scheme, each tuple is repartitioned exactly once. Furthermore, the precise estimation of the remaining time for joins essentially requires equi-weight histograms. In contrast to these adaptive approaches which rely on future load distribution estimation, we present equi-weight histograms that accurately capture workload skew and accordingly fairly partition the work. One could combine the two techniques to reap the benefits of both worlds. In particular, we can use our technique for initial partitioning and for feeding the estimator from [83] in the case of necessity for task reassignment. By doing so, we could obtain a scheme that adapts to run-time changes (e.g., network problems, machine failures), and that drastically reduces number of task reassignments compared to that of [83] alone.

Work-stealing. Work-stealing (e.g., [14]) is a concept related to adaptive load balancing, but with important differences. Rather than moving the partitions among the machines, it implies dividing the workload into many more partitions than the number of available machines. Each machine pulls a new partition once it finishes processing the previous one. However, increasing the number of partitions inherently increases replication. For example, if we divide a partition into two sub-partitions, the corresponding tuples from the opposite relation need to be duplicated. Thus, work-stealing increases the input-related work. Finally, it is not clear how to decide on the number of partitions so that work-stealing avoids JPS.

Sample data structures. The closest data structure to our sample matrix $\mathcal{M}_S$ is single-relation multi-attribute histogram [99, 113, 11, 35], which is used for selectivity estimation. These histograms represent the frequency distribution over a multi-dimensional space. In our $\mathcal{M}_S$, frequency is the number of output tuples for the corresponding segments of the input relations. However, the goal of multi-attribute histograms differs from ours as their aim is to minimize the total frequency errors over the entire domain, rather than to lend support for load balancing. Namely, multi-attribute histograms cannot provide for load balancing, as they capture the frequency rather than the weight distribution, and they cannot guarantee the maximum cell weight nor decide on the $\mathcal{M}_S$ size n_s. In addition, we take advantage of join peculiarities, that is, monotonicity.

4.6 Evaluation

This section compares our operator with state-of-the-art operators. We first evaluate the execution time and resource consumption, that is, memory requirements and network communication. Then, we assess the scalability of each operator. Further, we evaluate the accuracy of our partitioning scheme, along with the efficiency of building it. Finally, we analyze worst-case scenarios for our operator.

4.6.1 Experimental Setup

Environment. We perform our experiments on an Oracle Blade 6000 server with 10 Oracle X6270 M2 blades. Each blade has two 3Ghz 6-core Intel Xeon X5675 CPUs. Each blade runs Ubuntu 12.04 and has 72GB of DDR3 RAM and a 1Gbit Ethernet interface. Later on, by a machine assigned to an operator, we mean a core with an exclusively assigned portion of the blade main memory.

Datasets. We experiment on joins over both TPC-H [8] and a synthetic dataset X. We employ the TPC-H generator [44], which creates datasets with *Zipf* distributions set through the skew parameter z. We set $z = 0.25$ to demonstrate that JPS can be large even if RS is moderate. The X dataset has 2 independently generated relations (R_1 and R_2), each with 2 segments. The second and first segment sizes are in proportion 80/20, and joining smaller segments from R_1 and R_2 produces majority of the output. In particular, in the first segment, we generate x tuples and choose its join keys uniformly at random from the $[0..x/6]$ domain. In the second segment, we generate $y = 4 \cdot x$ tuples and choose its join keys uniformly at random from the $[2y, 6y]$ domain. The segments from different relations are independently generated.

Operators. We evaluate three different join partitioning schemes: (i) CI (1-Bucket scheme) [106], (ii) CS_I (M-Bucket scheme) [106], and finally, (iii) CS_{IO}, which is our equi-weight histogram scheme. We label the operator by the name of the employed partitioning scheme.

Configuration. We run the join queries on $J = 32$ machines, whereas for the scalability experiments we use 16 to 64 machines. For the joins over the TPC-H data, we set the scale factor to 160 (i.e., 160GBs) and for the scalability experiments, we set it between 80 and 320. In the histogram algorithm of CS_I[xvi], we set the number of buckets p to 2000. In the scalability experiments, we scale p proportionally to the number of machines J (e.g., $p = 4000$ for $J = 64$).

Programming model. We run experiments using our system SQUALL, which is based on STORM[xvii]. Storm is Twitter's backend engine for data analytics. In this chapter, we use Squall for offline processing. That is, we are interested only in the runtime of the entire query plan, rather than in per-tuple latencies. In that context, we can consider Squall as an in-memory

[xvi] This histogram algorithm does not create equi-weight histograms; it is a heuristic that builds a partitioning scheme.
[xvii] http://storm.apache.org/

Table 4.4: Joins' characteristics. $Input$ and $output$ sizes are in millions of tuples. β is the width of the band.

Name	Dataset	Join condition	$input$	$output$
B^*_{ICD}	TPC-H	Band-join($\beta = 2$)	480M	296M
B_{CB}	X	Band-join($\beta = 1$)	192M	348M
B_{CB}	X	Band-join($\beta = 3$)	192M	812M
B_{CB}	X	Band-join($\beta = 16$)	192M	3828M
BE^*_{OCD}	TPC-H	Band/Equi-join($\beta = 2$)	36.8M	2000M

* For joins over the TPC-H data, the database size is 160G and $z = 0.25$.

[xviii] MapReduce-like system, in which we partition the data and locally use indexes (hash or balanced binary tree), rather than sorting the data (which is typical for the map output in MapReduce systems). In the context of MapReduce, a join is defined in terms of map and reduce stages. Mappers shuffle the input tuples according to the partitioning scheme of the operator. Reducers perform the actual join and randomly shuffle the output tuples to the mappers of the next stage (e.g. join, aggregation). Thus, each mapper performs the same amount of work. Hence, to achieve global query plan load balancing, it suffices to balance the load among the reducers of a job. The job execution time includes the time of sending the tuples over the network to the next stage. We use up to 64 machines for the join, and up to 16 machines for the next stage. Squall runs in Java JRE v1.7.

Cost model. For our experiments we define the weight function (Section 4.2) for load balancing among the reducers as[xix]:

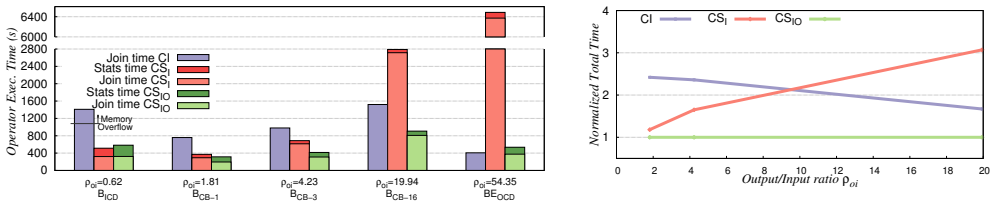
$$w(r) = c_i(r) + c_o(r) = w_i \cdot input(r) + w_o \cdot output(r)$$

where w_i, w_o is the average time cost of processing a single input and output tuple, respectively. We determine the values for w_i and w_o using linear regression on several benchmark runs. The regression method automatically divides all the communication and computation costs into c_i and c_o costs. The results of regression in our system suggest the values $w_i = 1$ and $w_o = 0.2$ for band-joins and $w_i = 1$ and $w_o = 0.3$ for combinations of equi- and band-joins.

Joins. As the operator performance is highly correlated to the $output/input$ cost ratio (c_o/c_i), we classify joins to *input-cost dominated (ICD)*, *cost-balanced (CB)* and *output-cost dominated (OCD)*. We evaluate a band-join (B_{ICD}) and a join with band and equality join conditions (BE_{OCD}) over the TPC-H dataset, and a band-join (B_{CB}) with 3 different widths of the band (β) over the X dataset. Table 4.4 summarizes the joins' characteristics. B_{ICD} is an input-cost dominated join, as $c_i = 8.1 \cdot c_o$. BE_{OCD} is an output-cost dominated join, as $c_i = 0.06c_o$. Finally, B_{CB} is cost-balanced, as the $input$ and $output$ cost are comparable ($0.25c_o \leq c_i \leq 2.75c_o$, depending on β). For the scalability experiments, we run the joins with the input of up to 480 million and the output of up to 8.8 billion tuples. The joins are defined in Section 4.7.3.

[xviii] See Section 4.4.2 for a discussion about alternative architectures.

[xix] The model can be flexibly adapted to represent any realistic cost function, as described in Section 4.2.4.



(a) Total Execution Time.

(b) Normalized Execution Time $B_{CB-\beta}$.

Figure 4.5: Execution times

4.6.2 Performance Analysis

In this section, we evaluate the operators' performance in terms of the execution time and resource consumption, that is, memory requirements and network communication. We show that the execution time mainly depends on the join $output/input$ ratio, which we call ρ_{oi}.

Total execution time is shown in Figure 4.5a as the sum of the time for building the partitioning scheme ("stats time") and the join execution time ("join time"). CI has only "join time" as it has no preprocessing phase, as it requires only the input sizes. Figure 4.5b shows the normalized total execution times for B_{CB}. The operators' execution times are highly correlated to the $output/input$ ratio ρ_{oi}: (i) For small ρ_{oi}, that is, on one side of the spectrum, $input$ costs dominate the join execution time. Thus, CI, which replicates each input tuple to 6 machines, performs poorly for B_{ICD}. CS_I avoids this problem, and as the join is input-cost dominated, the lack of output statistics and JPS do not affect performance. (ii) For high ρ_{oi}, that is, on the other side of the spectrum, $output$ costs dominate the join execution time. Thus, for BE_{OCD}, the effect of JPS on the join execution time of CS_I escalates, causing CS_I to perform poorly. CI achieves almost perfect load balancing for the $output$ costs by randomly assigning the tuples to the machines. High input replication does not affect performance for CI because it is an output-cost dominated join. (iii) As B_{CB} is a cost-balanced join, both existing operators perform worse than CS_{IO}. Increasing the width of the band β in B_{CB} leads to the increase in ρ_{oi}, such that the $output$ costs grow relatively to $input$ costs. This improves the performance of CI and degrades the performance of CS_I compared to CS_{IO}.

Our CS_{IO} outperforms the other operators as it is close-to-optimum on the *total* work per machine, which includes both $input$ and $output$ costs. CS_{IO} captures the output distribution *and* avoids high input tuple replication. Thus, in terms of the join execution time, CS_{IO} achieves from 1.01× slowdown (B_{ICD}) to 16.99× speedup (BE_{OCD}) compared to CS_I, and from 1.08× (BE_{OCD}) to 4.36× (B_{ICD}) speedup compared to CI. In terms of the total execution time, CS_{IO} achieves up to 12.12× (BE_{OCD}) speedup compared to CS_I, and up to 2.42× (B_{ICD}) speedup compared to CI. In fact, as CI for B_{ICD} runs out of memory, we extrapolate its total execution time using the cost model parameters and the percentages of processed input and output tuples. Note that in terms of total execution time, CS_{IO} performs worse than the best among CI and CS_I at the extremes of the spectrum. Namely, for B_{ICD}, CS_{IO} is 1.13× slower than CS_I. Similarly, for BE_{OCD}, CS_{IO} is 1.31× slower than CI. This is because CI does not

Table 4.5: Join execution and histogram algorithm time (s) of CS_I for different number of buckets p.

Join	Time	Number of buckets p					
		2000	4000	8000	10000	16000	24000
BE_{OCD}	Join execution	6372	6306	5480	5080	4294	3410
	Histogram alg.	0.4	1.3	5	8.1	19	49
B_{CB}	Join execution	615	604	601	582	569	575
$\beta = 3$	Histogram alg.	0.4	1.4	4.9	6.7	15	36

incur any time for building the partitiong scheme, and building the partioning scheme is more involving for CS_{IO} than for CS_I.

All the SQUALL operators will be faster once we migrate from STORM to Twitter HERON, as HERON is an order-of-magnitude faster[xx] than STORM. We describe HERON's design choices that lead to performance improvements in Section 2.3. HERON is API-compatible with STORM. At the time of preparing this thesis, HERON was not open source yet. In the meantime, HERON became an open source project, and we are currently porting SQUALL to HERON. Once we complete the porting, SQUALL will be on par with other parallel main-memory database systems (e.g. Track Join [111]).

Choosing among CS_I and CI. An important difficulty when using the existing CS_I and CI schemes is that we cannot always choose the better one without knowing the input/output sizes. However, output size estimation using the precomputed statistics is error-prone due to possible predicate correlations [74]. In fact, the estimate can be orders of magnitude away [122]. Output size estimation is even harder for non-equi joins. *This might lead to choosing the worst operator among CI and CS_I* (e.g. CI for an input-cost dominated join). In that case, our CS_{IO} achieves from 2.42× (B_{ICD}) to 12.12× (BE_{OCD}) speedup.

More detailed input statistics in CS_I (increased number of buckets p in CS_I) cannot cure the lack of output statistics nor address JPS. In particular, Table 4.5 shows that for both BE_{OCD} and B_{CB-3}, increasing p leads to increased histogram algorithm time, and thus, increased time for building the partitioning scheme. Increasing p also decreases the join execution time. However, even if CS_I is given more time for building the partitioning scheme than CS_{IO}, its total execution time is still much worse than that of CS_{IO}. For instance, for BE_{OCD} and $p = 24000$, the histogram algorithm grows to 49 seconds, making the time for building the partitioning scheme on par with that of CS_{IO}. Moreover, in that case, the total execution time of CS_I is 6.67× worse than that of CS_{IO}.

Resource consumption. Figure 4.6 illustrates resource consumption, which includes cluster memory and network traffic (input data sent from mappers to reducers). Resource consumption is important because it directly translates to energy consumption and, in cloud environments, to dollar costs. In general, CI has better execution times than CS_I, but it is very

[xx] http://www.infoq.com/news/2015/06/twitter-storm-heron

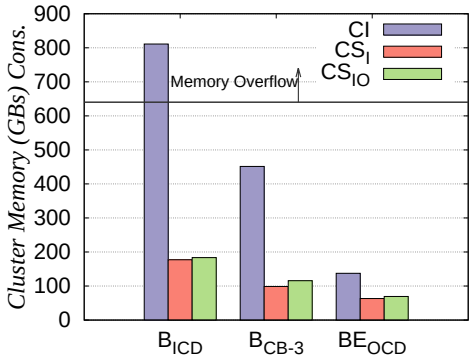


Figure 4.6: Cluster Memory Consumption.

resource-inefficient compared to both CS_I and CS_{IO}.

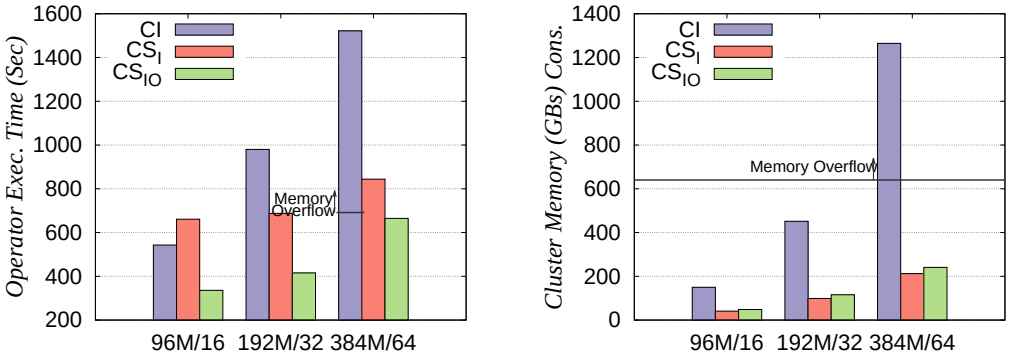
For B_{ICD} and B_{CB-3}, CI requires around 4× more memory than CS_I and CS_{IO}. This is due to the fact that CI excessively replicates tuples (the replication factor is 6, as the partitioning scheme is 4×8 so one relation is replicated 4× and the other is replicated 8×). CI requires 4× rather than 6× more memory as both CS_I and CS_{IO} replicate some input tuples (see Figures 4.1c, 4.1d). CS_{IO} uses slightly more memory than CS_I, because CS_{IO} balances on the total work, so it assigns more $input$ for the regions with relatively small $output$. In fact, as CI for B_{ICD} runs out of memory, we extrapolate its memory consumption using the percentage of the processed input tuples. CI in BE_{OCD} does not have high memory consumption, as the input size is smaller than for B_{ICD} and B_{CB-3}.

4.6.3 Scalability

Next, we evaluate the weak scalability of the operators by scaling the data size and the number of machines evenly. We show that our CS_{IO}, in contrast to CS_I and CI, scales well both in the total execution time and resource consumption.

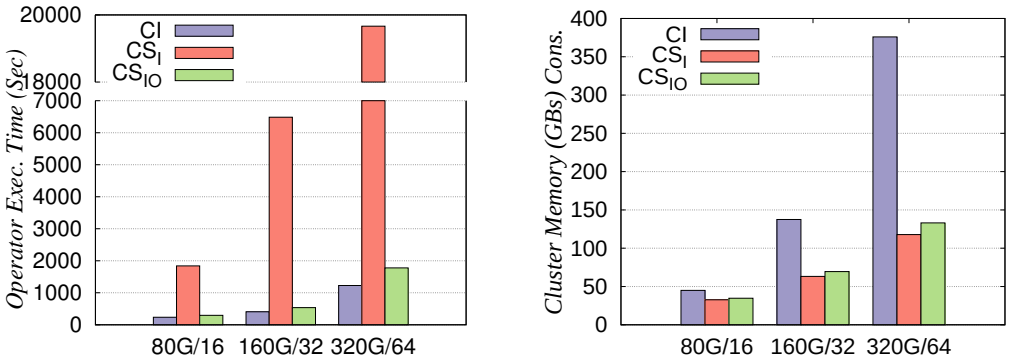
B_{CB-3} total execution time is shown in Figure 4.7a. We evaluate various $input/output/J$ settings, more specifically, $96M/406M/16$, $192M/811M/32$ and $384M/1.62B/64$, where M and B stand for millions and billions of tuples, respectively. CI scales worse than CS_I and CS_{IO}. This is expected, as the replication factor grows from 4 ($J = 16$) to 8 ($J = 64$), which doubles the $input$ costs on each machine. Namely, for $J = 16$ and $J = 64$, CI has 1.62× and 2.29× worse total execution time than CS_{IO}, respectively. In fact, as CI with $J = 64$ runs out of memory, we extrapolate its total execution time and memory consumption.

The join execution time for CS_{IO} grew from 310s ($J = 32$) to 479s ($J = 64$). Given that the work per machine did not grow between $J = 32$ and $J = 64$, the only possible explanation would be reaching the shared network limits. In fact, the network throughput per machine for $J = 32$ is more than twice as high compared to $J = 64$ (20MB/s vs only 7.5MB/s, respectively).



(a) B_{CB-3} Scalability Execution Time. (b) B_{CB-3} Scalability Memory Cons.

Figure 4.7: Scalability of B_{CB-3}.



(a) BE_{OCD} Scalability Execution Time. (b) BE_{OCD} Scalability Memory Cons.

Figure 4.8: Scalability of BE_{OCD}.

***B_{CB-3}* resource consumption.** Figure 4.7b shows the cluster memory consumption for B_{CB-3}. As the replication factor grows from 4 ($J = 16$) to 8 ($J = 64$), CI requires increasingly more memory compared to CS_I and CS_{IO}. Namely, for $J = 16$ and $J = 64$, CI requires 3.1× and 5.25× more memory than CS_{IO}, respectively. As CI with $J = 64$ runs out of memory, we extrapolate its memory consumption.

***BE_{OCD}* total execution time** is shown in Figure 4.8a. The $input/output/J$ settings are $21.2M/612M/16$, $36.8M/2B/32$ and $62M/8.8B/64$. Taking into account that increasing J from 16 to 64 (4×) causes the output size to grow 14.46×, CS_{IO} and CI achieve good scalability. For CS_{IO}, this validates the efficiency of our scheme even for highly output-cost dominated joins (for $J = 64$, $\rho_{oi} = 142.57$). For CI, this is due to the fact that the $output$ cost outweighs the $input$ cost. On the other hand, CS_I scales very poorly as JPS causes that only few machines produce most of the output. Namely, for $J = 16$, $J = 32$ and $J = 64$, CS_I has 6.23×, 12.12× and 11.06× longer total execution time than CS_{IO}, respectively.

For $J = 64$, CS_{IO} performs 1.45× worse than CI. This is primarily due to the time for building

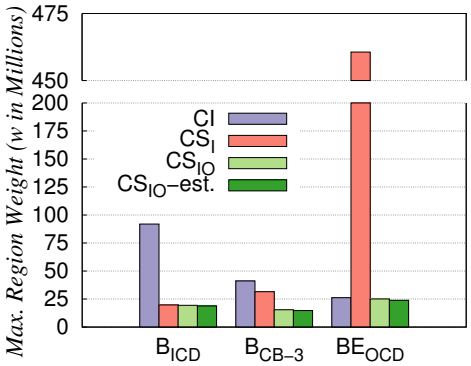


Figure 4.9: Maximum Region Weight.

the CS_{IO} partitioning scheme, which took 531s for $J = 64$. We identified several ways to improve the CS_{IO} stats time, but we leave them for future work. For now, if needed, we can resort to fall back approach described in Section 4.6.5. The CS_{IO} join execution time (1246s) is comparable to the total CI time (1227s). This is because BE_{OCD} is an output-cost dominated join, and CI achieves perfect load balancing for the output.

***BE_{OCD}* resource consumption.** Figure 4.8b shows the cluster memory consumption for BE_{OCD}. The gap between CI and the other two operators is smaller than in B_{CB-3} due to the following. As the number of machines J grows from $J = 16$ to $J = 64$ (4×), the input size grows only 2.92×. Thus, for $J = 64$, CI takes 2.82× more memory than CS_{IO}.

Scalability summary. Overall, in terms of the total execution time and resource consumption, only CS_{IO} scales well for both B_{CB-3} and E_{OCD}.

4.6.4 Accuracy and Efficiency of CS_{IO}

This section evaluates the accuracy of our partitioning scheme, as well as the efficiency of building it. Building the partitioning scheme consists of collecting the input and output samples and running our 3-stage histogram algorithm.

Accuracy of the cost model. Our cost model represents the join work of a machine as the weight of the region assigned to it. In a parallel setting, the machine assigned the biggest amount of work determines the join execution time. Thus, the cost model is accurate if the region weight corresponds to the machine's work, that is, if the maximum region weight corresponds to the join execution time. Figure 4.9 validates the model accuracy as for each join among B_{ICD}, B_{CB-3} and BE_{OCD} ($J = 32$), the maximum region weights of different schemes are proportional to the corresponding join execution times. The proportionality holds only within the same join, as a weight unit represents different amount of work in different joins. We obtain the weights after the join execution by computing the weight function on the number of input and output tuples per machine.

Accuracy of our CS_{IO}. Figure 4.9 shows that the estimated maximum region weight in our histogram algorithm, marked as CS_{IO}-EST., is at most 6% off the value computed after the execution. Thus, our scheme is accurate.

The time for building the partitioning scheme is illustrated in Figure 4.5a as "stats time". It includes the time to collect statistics (the input statistics for CS_I and input and output statistics for CS_{IO}), and the running time of the histogram algorithm. We evaluate the efficiency of building our partitioning scheme, and compare it to that of CS_I.

Figure 4.5a shows that building the CS_{IO} scheme takes at most 44.5% of its total execution time (B_{ICD}). Furthermore, building the CS_{IO} scheme is at most 11.7% more expensive than that of CS_I in terms of the CS_{IO} total execution time (B_{ICD}). This is due to the following reasons: (i) Collecting the input statistics is much cheaper in CS_{IO} than in CS_I. CS_I requires 2 MapReduce stages, while CS_{IO} requires only 1 MapReduce stage. This is due to the fact that CS_I needs to use more buckets than CS_{IO} to account for the error caused by the lack of the output statistics. In the worst case (high JPS), the required number of buckets for CS_I is $\Theta(n)$. In contrast, the number of buckets for CS_{IO} does not depend on skew at all, and it slowly grows with the increase in n ($O(\sqrt{n})$). (ii) The time to collect output statistics for CS_{IO} is not much longer than the second pass of collecting input statistics for CS_I. In addition to a scan over the input relations, which is required by both operators for building the partitioning scheme, CS_{IO} performs a scan over d_{2equi} (step 2 in Section 4.4.1) and produces the output sample (step 3 in Section 4.4.1). The former is cheap as d_{2equi} tends to be much smaller than its originating relation, which is the smaller one. The latter is cheap as the output sample size is very small compared to n ($s_o = \Theta(\sqrt{nJ})$).

Accuracy/Efficiency summary. Overall, our equi-weight histogram scheme is practical, as it provides for both accurate and efficient load balancing.

4.6.5 Sensitivity analysis

Worst-case scenarios

Input-cost dominated joins (small ρ_{oi})/no skew. For very small ρ_{oi} (B_{ICD}), CS_{IO} achieves minimal slowdowns in the join execution time compared to CS_I (1.01×). This is because the $output$ cost is 8.1× smaller than the $input$ cost, so JPS minimally affects the performance. In fact, joins with very small ρ_{oi} behave almost as if there was no JPS at all. Thus, in the worst case (B_{ICD}), the total execution time of CS_{IO} is 1.13× higher than that of CS_I. This is acceptable compared to the speedups that our scheme achieves for other joins.

High-selectivity joins (very high ρ_{oi}). Our scheme is designed for low-selectivity joins. CS_{IO} is better or on par with CI if the output is up to 2 orders of magnitude bigger than the input. We address high-selectivity joins as follows. As we saw in $Section$ 4.6.2, we cannot know join selectivity beforehand. Rather, we take advantage of the following fact. As ρ_{oi} grows, building

CS_{IO} requires less time compared to the total execution time of the better among CI and CS_{IO} (from 44.5% for B_{ICD} to 29.9% for BE_{OCD}, see Figure 4.5a). This is because higher ρ_{oi} implies producing more output tuples and leads to more join work. On the other hand, when building the partitioning scheme, we need to produce only a small sample of the join output. As building the CS_{IO} scheme is comparably cheap for high-selectivity joins, we always start with our scheme, and fall back to CI if needed. We decide when to switch to CI using the following fact. As ρ_{oi} grows, the time for building the CS_{IO} scheme grows relatively to the input sizes (259s for 480M tuples of B_{ICD} to 160s for only 37M tuples of BE_{OCD}). If the time for building the scheme exceeds a certain experimentally-found threshold (e.g. one second for each million of input tuples in our setup), we fall back to CI. In that case, we waste only 8% of the total execution time of CI before switching to CI.

Next, we analyze the difference in memory consumption between CS_{IO} and CI as a function of ρ_{oi}. For small ρ_{oi} (B_{ICD}), CI takes 4.42× more memory than CS_{IO} (see Figure 4.6). Whereas, for large ρ_{oi} (BE_{OCD}), CI takes 1.98× more memory than CS_{IO}. As we move more to the right of the ρ_{oi} spectrum, the savings in memory of CS_{IO} compared to CI diminish. Interestingly, CI starts to outperform CS_{IO} (for ρ_{oi} = 54.35 and BE_{OCD}) when it still takes more memory (1.98×) than CS_{IO}. This illustrates the tradeoff between total execution (which includes building the partitioning scheme) and memory consumption.

Summary. In the worst case of data distribution (input-cost dominated or high-selectivity joins) CS_{IO} can perform worse (up to 1.13× for B_{ICD}, up to 1.31× for BE_{OCD}, $J = 32$ and up to 1.45× for BE_{OCD}, $J = 64$) than the better among the existing schemes. For low ρ_{oi}, CS_I outperforms CS_{IO} as the latter collects the output sample, while JPS makes very little difference in performance. In other words, for small ρ_{oi}, the sampling overheads are considerable compared to the join processing time. For instance, building the partitioning scheme for B_{ICD} for CS_{IO} is 80% of the corresponding join execution time (44.5% of the corresponding total execution time), as Figure 4.5a shows. For high ρ_{oi}, CS_{IO} falls back to CI, and the overhead is due to the time taken before deciding to switch to CI. CI is a better choice for high-selectivity joins as it achieves almost perfect load balancing on both input and output, without requiring the input or output sample nor running a histogram algorithm.

Overall, possible slowdowns due to overheads of using our partitioning scheme (up to 1.45×) are acceptable compared to the achieved speedups (up to 12.12×). When the fall back approach is used, the slowdowns are only up to 1.13×. Finally, CS_{IO} subsumes both CI and CS_I in a sense that CS_{IO} often performs better than the better among CI and CS_I schemes. This is because CS_I addresses only RS, and CI addresses both RS and JPS but with high replication cost. Our CS_{IO} addresses both RS and JPS, while minimizing tuple replication.

Discussion

The importance of JPS. The importance of JPS is directly connected to the $output/input$ ratio which we call ρ_{oi}. From Figure 4.5a, we can see that JPS becomes more important (CS_{IO}

performs better compared to CS_I) as ρ_{oi} grows. On the other hand, for joins with small ρ_{oi}, that is, for joins with relatively small output (e.g., B_{ICD} from Figure 4.5a), JPS minimally affects performance. This is effectively equivalent to execution without any JPS, and in that case CS_I may outperform our CS_{IO}.

The importance of JPS also depends on the processing costs. In our setup, the output-related work for joins is significant, especially when join output is considerably large. This is due to sending the join output over the network. In a different setting, when performing a directed join, the output tuples stay in the memory of the machine which produced them (e.g., Track-Join [111]). In that case, the next operator in the query plan takes into account initial data distribution, and produces the join output while minimizing the number of tuples transfered over the network. In such an environment, the cost of processing an output tuple is much smaller than the cost for processing an input tuple, and JPS does not affect performance much, even if ρ_{oi} is relatively high.

If JPS has a minimal effect on the performance due to any of the reasons mentioned above, we recommend using the CS_I scheme, as it does not collect the output sample. Furthermore, for low-selectivity joins without JPS (i.e., uniform output distribution over the join matrix), we expect CS_{IO} to be on par with CS_I (for small ρ_{oi}, CS_{IO} may be slightly slower due to collecting the output sample). In that case, the benefit of using our CS_{IO} is that our scheme subsumes both CS_I and CI, and the performance difference between CS_I and CI is often significant (so choosing the wrong operator among the existing ones affects performance considerably).

Relative relation sizes. The performance difference between a content-insensitive (CI) and a content-sensitive scheme (CS_I or CS_{IO}) depends on the relative relation sizes. Let us consider a general case of two input relations R_1 and R_2, where $|R_2| = x \cdot |R_1|$. Given J machines, the optimum CI partitioning is $J_1 \cdot J_2$, where $J = J_1 \cdot J_2$ and $J_2 = x \cdot J_1$ (for more details on the analysis, please refer to the Random-Hypercube scheme from Section 5.2, as CI is a 2-dimensional Random-Hypercube). The load per machine for the CI scheme is:

$$L_{CI} = \frac{|R_1|}{J_1} + \frac{|R_2|}{J_2} = \frac{2 \cdot |R_1|}{J_1} \tag{4.1}$$

Next, we analyze the load per machine for a content-sensitive scheme. The exact load depends on the data distribution and the concrete scheme used. We simplify the analysis by computing the lower bound on the load per machine, which implies uniform distribution (no skew) and no input tuple replication. Given the same number of machines J as for a content-insensitive scheme, we express the parallelism as $J = J_1 \cdot J_2$ and $J_2 = x \cdot J_1$, in order to have the same variables as in Equation 4.1. Thus, the formula for a content-sensitive scheme is:

$$L_{CS} = \frac{|R_1| + |R_2|}{J} = \frac{|R_1| \cdot (1 + x)}{x \cdot J_1^2} \tag{4.2}$$

Hence, the upper bound on the improvement in load per machine that a content-sensitive

scheme can achieve over a content-insensitive scheme is:

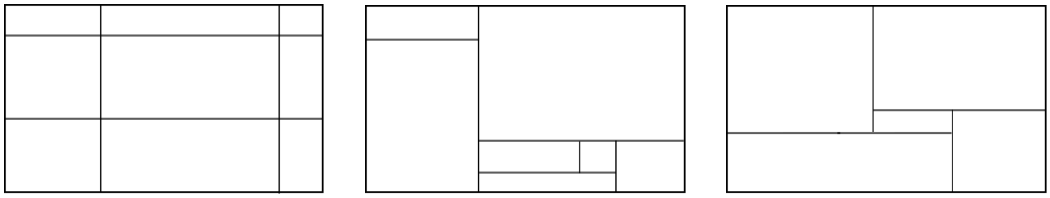
$$\frac{L_{CI}}{L_{CS}} = \frac{2 \cdot x \cdot J_1}{1 + x} \tag{4.3}$$

For example, if $J = 64$ and both relations are of the same size ($J_1 = J_2 = 8$ and $x = 1$), $\frac{L_{CI}}{L_{CS}} = 8$. On the other hand, if $J = 64$ and $|R_2| = 64 \cdot |R_1|$ ($x = 64$, $J_1 = 1$, $J_2 = 64$), $\frac{L_{CI}}{L_{CS}} \approx 2$. Overall, the potential memory savings and performance speedup when using a content-sensitive scheme is much higher for relations of comparable sizes. This analysis is in accordance with a common strategy which employs a broadcast join (a special case of the CI scheme) when one relation is much smaller than the other one. Given that CI achieves perfect load balancing on output while only requiring relative relation sizes, it is the best choice among CI, CS_I and CS_{IO} when one input relation is considerably smaller than the other one.

The number of candidate cells and their detection. The content-sensitive schemes (such as CS_I and our CS_{IO}) are designed for low-selectivity joins, where only a small portion of the join matrix produces output tuples. To that end, we categorize the join matrix cells into candidates (potentially producing output tuples) and non-candidates (guaranteed to not produce any output). Low selectivity refers to the proportion between the number of the output and input tuples in the join, but also to the number of candidate cells. There are scenarios when the output is comparable to the input (satisfying the first requirement for low selectivity joins), but it is dispersed across the entire join matrix. Thus, majority of the matrix cells are candidates, and the histogram algorithm (both for CS_I and our CS_{IO}) becomes very expensive. In that case, CI achieves the best performance.

In addition to having a relatively small number of candidate cells, the content-sensitive schemes (CS_I and our CS_{IO}) need to quickly identify whether a join matrix cell is a candidate or not. Some join conditions allow candidacy-checking according to the cell boundary keys (e.g., a band-join). If the only way to find candidates is to examine all the tuples within the matrix cell, this becomes equivalent to join execution, defeating the purpose of finding candidates and building the partitioning scheme. In that case, using the CI is the best option.

Join conditions. We focus on monotonic joins, which include combinations of equi-joins, band joins and inequality joins. A 2-way join which includes one or more equality join conditions and a band join condition frequently occurs in practice. Interestingly, we can turn this join into an equivalent one which has only a band join condition with a join key which is a concatenation of the different join keys from the original join. An alternative way to execute this join is to partition the input relations using the equality join conditions (e.g., using the scheme from [29]), and then locally perform additional filtering using the band join condition. Such an execution plan in the case of uniform output distribution might execute faster than the one using CS_{IO} or CS_I over the entire join, as it does not require running a histogram algorithm for building the partitioning scheme. However, such an execution plan is prone to JPS. In particular, although different machines receive roughly the same number of input tuples, the number of output tuples sent over the network for the next operator in the query



(a) Grid partitioning (b) Hierarchical partitioning (c) Arbitrary partitioning

Figure 4.10: Types of partitionings.

plan (or written to disk) varies due to join selectivity variations of the band join. In contrast, CS_{IO} addresses JPS, and its performance is more reliable (performance reliability is defined in Section 4.5).

4.6.6 Summary

Joins are defined in a spectrum of cost distribution. At each end, either $input$ or $output$ costs dominate the join cost. Previous work, that is, CS_I and CI, perform well only at the extreme ends of the $output/input$ spectrum. This is because CI suffers from excessive input tuple replication (which worsens with the increase in the number of joiners), while CS_I cannot capture the $output$ cost distribution. Due to errors in the output size estimation, especially for non-equi joins, choosing the wrong operator among CS_I and CI becomes plausible, causing severe performance degradations. In contrast to previous work, our CS_{IO} captures the output distribution, and avoids high input tuple replication. Thus, our scheme is close-to-optimum on the *total* work per machine, which includes both $input$ and $output$ costs. Consequently, our scheme performs very well over a wide spectrum of $output/input$ ratios, and it scales with increasing data sizes.

CS_{IO} achieves up to 5.25× improvement in resource consumption (for B_{CB-3}, $J = 64$) and up to 2.42× speedup (for B_{ICD}, $J = 32$) compared to CI. Moreover, CS_{IO} achieves up to 12.12× speedup compared to CS_I (for BE_{OCD}, $J = 32$). As these speedups refer to the total execution time, they also validate the efficiency of building our scheme, which consists of collecting the input and output samples and running our 3-stage histogram algorithm.

4.7 Further details

4.7.1 Types of partitioning

Figure 4.10 shows 3 different types of partitioning from the computational geometry literature [101, 102]. Grid partitioning results from dividing the original rectangle into rows and columns. Hierarchical partitioning is generated by recursively dividing the original rectangles into 2 subrectangles using a horizontal or vertical line. Arbitrary partitioning allows any partitioning to rectangles.

4.7.2 The histogram algorithm: Details and proofs

Sampling

The following lemmas require that $w(r)$ is *monotonic* [xxi] (a region weighs at least the weight of any of its subregions) and that $c_i(r)$ and $c_o(r)$ are *superadditive*, that is, processing $x+y$ input (output) tuples is at least as expensive as the sum of processing costs for x and y input (output) tuples. This holds for realistic cost models.

Lemma 4.3.1. *$n_s = \sqrt{2nJ}$ is the minimum $\mathcal{M}_S$ size such that the maximum cell weight σ in $\mathcal{M}_S$ is at most half of the maximum region weight of the optimum $\mathcal{M}_\mathcal{H}$ partitioning. This holds independently from the join condition and the join key distribution, given that $m \geq n$* [xxii].

Proof. An $\mathcal{M}_S$ cell corresponds to a region in the original matrix with dimensions $n/n_s \times n/n_s$, where n_s is the number of buckets in the equi-depth histogram. Due to the fact that $n_s = \sqrt{2nJ}$, the semi-perimeter of each cell is $\max_{cell}(s_p(cell)) = 2n/n_s = \sqrt{2n/J}$. The maximum frequency of an $\mathcal{M}_S$ cell is given by the Cartesian product between the encompassed input tuples from the two relations. That is, $\max_{cell}(f(cell)) \leq (n/n_s)^2 = n/2J$. Because $m \geq n$, it follows that $\max_{cell}(f(cell)) \leq m/2J$. It holds that $\sigma = \max_{cell}(w(cell)) \leq c_i(\sqrt{2n/J}) + c_o(m/2J)$. As $J \ll n$ (it suffices that $J < n/3$), it follows that $\sqrt{2n/J} < n/J$ and $\sigma \leq c_i(n/J) + c_o(m/2J)$. We denote the maximum region weight of the $\mathcal{M}_\mathcal{H}$ optimum partitioning as w_{OPT}. It holds that $w_{OPT} \geq (c_i(2n) + c_o(m))/J$, as each incoming tuple is assigned to at least one region. Since c_i and c_o are superadditive, it follows that $w_{OPT} \geq c_i(2n/J) + c_o(m/J)$ and that $\sigma \leq w_{OPT}/2$. More precisely, as the bucket sizes are probabilistic, we can conclude that with high probability, these bounds are very close to the actual bounds. □

For the next lemma, we will need to define input and output sample sizes.

Input sample size s_i. For each relation, we build an approximate equi-depth histogram [42]. Namely, for each relation, we take a random uniform sample of size s_i, sort the sampled tuples according to the join key, and then build an equi-depth histogram on them with $n_s < s_i$ buckets.

According to [42], for a given n_s, s_i needs to be at least $4n_s \ln(2n/\gamma)/e^2$, where e is the maximum error on the bucket size with probability of at least $1-\gamma$. This implies that a small sample of size $s_i = \Theta(n_s \log n)$ is sufficient for building approximate equi-depth histogram.

Output sample size s_o. In [99], the authors show that the sample size is not a function of the actual data size, and that it can be obtained from standard tables based on Kolmogorov's statistics [64]. For example, for an error on the region $output$ within 5% and confidence of at least 99%, the standard tables only require that the sample size is at least 1063. On the

[xxi] Monotonicity on the join output and monotonicity on the weight function are different and should not be confused.

[xxii] This typically holds in practice. We relax it later in this section.

other hand, the sample size should be a small integer multiple of the number of scrutinized categories in the population. In our case, this number is the number of candidate $\mathcal{M}_S$ cells (n_{sc}), as the non-candidate cells never produce an output tuple. Thus, the output sample size is $s_o \geq \max(1063, n_{sc})$ for the specified error margin and confidence interval. Consequently, we need an output sample of size $s_o = \Theta(n_{sc})$.

To determine n_{sc}, we define a low-selectivity join precisely. As Section 4.2.2 states, the content-insensitive (CI) scheme achieves (almost) perfect load balancing for $output$. Thus, if the output size $m > \rho_B n$ ($\rho_B \gg 1$ is a constant), CI works very well. Hence, it pays off to use a content-sensitive scheme only if $m \leq \rho_B n$ [xxiii] , that is, if $m = \mathcal{O}(n)$. This condition defines a *low-selectivity* join. We require a similar condition to hold between the input and output of the sample matrix[xxiv]:

$$n_{sc} = \mathcal{O}(n_s) \tag{4.4}$$

Thus, we need a small output sample: $s_o = \Theta(n_s) = \Theta(\sqrt{nJ})$.

We next prove that, given this $n_s = \sqrt{2nJ}$ from Lemma 3.1, the sampling stage time complexity is low.

Lemma 4.3.2 [Extended version]. *The total sample size collected for building $\mathcal{M}_S$ is $\Theta(n_s \log n)$. The sampling stage running time is $\mathcal{O}(n_s \log n_s)$. For $n_s = \sqrt{2nJ}$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [xxv], the sample size and the time complexity are both $\mathcal{O}(n/J)$.*

Proof. From $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$, it follows that

$$\log n = \mathcal{O}(\sqrt{n/J^3}) \tag{4.5}$$

As the input sample size is $s_i = \Theta(n_s \log n)$, and the output sample size is $s_o = \Theta(n_s)$, the total sample size is dominated by the input. By substituting $\log n$ from Equation 4.5, the total sample size is $s_i = \Theta(n_s \log n) = \Theta(\sqrt{nJ} \log n) = \mathcal{O}(n/J)$.

Let us discuss the time complexity for building $\mathcal{M}_S$. To create approximate equi-depth histogram on input, we need to sort the input sample tuples. We do it on the sites providing the samples, incurring $\mathcal{O}(\log^2 s_i) = \mathcal{O}(\log^2 (n/J))$ time.

For each sample output tuple (s_o of them), we use binary search to find a $\mathcal{M}_S$ cell to increment. Thus, processing the output takes $\mathcal{O}(s_o \log n_s) = \mathcal{O}(n_s \log n_s)$ time. As $n_s \leq n$, it follows that

[xxiii] In our setup, our scheme works well even if m is two orders of magnitude bigger than n (see Section 4.6.3). Thus, we cover a wide range of joins in practice.

[xxiv] If any of these assumptions do not hold, we fall back to the content-insensitive operator (see Section 4.6.5 for details). However, we experimentally show that the assumptions hold for many interesting joins.

[xxv] See footnote vii.

$\log n_s \leq \log n$, and from Equation 4.5, $\log n_s = \mathcal{O}(\sqrt{n/J^3})$. Hence, $\mathcal{O}(n_s \log n_s) = \mathcal{O}(n/J)$. Thus, the total time complexity for building $\mathcal{M}_\mathcal{S}$ is bounded by $\mathcal{O}(n/J)$. □

Coarsening

The coarsening algorithm [102] is the RTILE (rectangle tiling) problem with grid $(n_c \times n_c)$ partitioning and the MAX-WEIGHT-ID metric. This is an approximation algorithm with an approximation ratio of 2 [102]. That is, given $\mathcal{M}_\mathcal{S}$ and n_c (the size of $\mathcal{M}_\mathcal{C}$), where the maximum $\mathcal{M}_\mathcal{C}$ cell weight of the optimum grid partitioning is ϕ_0, the algorithm returns an $\mathcal{M}_\mathcal{C}$ with maximum cell weight $\phi \leq 2\phi_0$.

For sparse matrices, if range trees are used for computing the prefix sum, the coarsening algorithm [102] runs in

$$\mathcal{O}(n_{sc} \log n_{sc} + (n_s + n_c^2 \log n_{sc}) \cdot n_c \log n_s) \tag{4.6}$$

time, where n_s is the $\mathcal{M}_\mathcal{S}$ size, and n_{sc} is the number of candidates in $\mathcal{M}_\mathcal{S}$.

MonotonicCoarsening. Monotonicity (consecutiveness of candidate cells) allows us to visit all the n_{cc} candidate cells in $\mathcal{O}(n_{cc})$ time. Along the lines of Equation 4.4, we assume $n_{cc} = \mathcal{O}(n_c)$. Consequently, the coarsening algorithm for monotonic joins performs only $\mathcal{O}(n_c)$, rather than n_c^2 weight computations per iteration. The complexity from Equation 4.6 then becomes:

$$\mathcal{O}(n_{sc} \log n_{sc} + (n_s + n_c \log n_{sc}) \cdot n_c \log n_s) \tag{4.7}$$

Lemma 4.3.3. *The running time of the coarsening algorithm is* $\mathcal{O}((n_s + n_c^2 \log n_s) \cdot n_c \log n_s)$. *For* $n_c = 2J$ *and* $J = \mathcal{O}(\sqrt[3]{n/\log^2 n_s})$ [xxvi], *the time complexity becomes* $\mathcal{O}(n)$.

Proof. Equation 4.6 shows the total running time for building the coarsened matrix. By substituting $n_{sc} = \mathcal{O}(n_s)$ (Equation 4.4), $\log n_s$ from $J = \mathcal{O}(\sqrt[3]{n/\log^2 n_s})$, $n_c = \Theta(J)$ and $n_s = \Theta(\sqrt{nJ})$ into Equation 4.6, it follows that the complexity is $\mathcal{O}(n)$. □

Regionalization

Regionalization is an RTILE problem with arbitrary partitioning [101] and the MAX-WEIGHT-ID metric. There exist algorithms for this problem in the restricted, output-only case, e.g. [31]. However, they are not applicable for the general case, which entails support for: *a*) monotonic metrics (including the weight function) and *b*) segments which may or may not be covered by a region (0-cells). By design, these algorithms generate prolate regions and thus incur excessive $input$ costs. Hence, they can be arbitrarily worse in weight than the optimum. The best algorithm which works for the general case is Binary Space Partition (BSP) [30, 101].

[xxvi] See footnote vii.

Next, we prove lemmas from Section 4.3.3.

Lemma 4.3.4. *A rectangle is defined by the upper left and the lower right corner. For monotonic joins, each defining corner of a minimal candidate rectangle is a candidate cell, yielding $\mathcal{O}(n_c^2)$ minimal candidate rectangles in total.*

Proof. We prove the lemma using contradiction by assuming that a defining corner of a minimal candidate region is not a candidate cell. Let us consider the position of the upper left corner. If it is before the first candidate cell in the row of $\mathcal{M}_C$, the left boundary of the rectangle is empty (see rectangles r_1 and r_{min1} in Figure 4.4b). Thus, the rectangle is not a minimal candidate. If the position of the upper left corner is after the last candidate cell in the row, the upper boundary of the rectangle is empty (see rectangles r_2 and r_{min2} in Figure 4.4b). Again, the rectangle is not a minimal candidate. Consequently, the upper left corner must be a candidate cell. The proof for the lower right corner is symmetric. Thus, both defining corners of a minimal candidate rectangle are candidate cells.

Consequently, there are n_{cc}^2 minimal candidate rectangles, where n_{cc} is the number of candidate cells in $\mathcal{M}_C$. Along the lines of Equation 4.4, we assume $n_{cc} = \mathcal{O}(n_c)$. Thus, there are $\mathcal{O}(n_c^2)$ minimal candidate rectangles in total. □

Lemma 4.3.5. *The regionalization stage based on* MONOTONICBSP *runs in $\mathcal{O}(n_c^3 \log n_c \log n)$ time. For $n_c = 2J$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$, the stage takes $\mathcal{O}(n)$ time.*

Proof. Generating r_N minimal candidate rectangles and sorting them takes $\mathcal{O}(r_N \log r_N)$ time. Then, for each rectangle (there are r_N of them), we: *a*) compute its weight which takes $\mathcal{O}(1)$ time with $\mathcal{O}(n_c^2)$ prefix sum precomputation (line 16) and *b*) for each splitter line within a rectangle, $\mathcal{O}(n_c)$ of them, for both subrectangles, find the corresponding minimal candidate rectangle (using binary search it takes $\mathcal{O}(\log n_c)$ time) (lines 19-23). Step *b* yields $\mathcal{O}(n_c \log n_c)$ time per rectangle. Overall, this requires a total time of $\mathcal{O}(n_c^2 + r_N(\log r_N + n_c \log n_c))$.

From Lemma 4.3.4, we know that $r_N = \mathcal{O}(n_c^2)$. Thus, MONOTONICBSP runs in $\mathcal{O}(n_c^3 \log n_c)$ time. Due to transformation from DRTILE to RTILE, the regionalization stage takes $\mathcal{O}(n_c^3 \log n_c \log n)$ time. Given $n_c = \Theta(J)$, $\log J \leq \log n$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$, it follows that the stage takes $\mathcal{O}(n)$ time. □

Putting it all together

Theorem 4.3.1 [Extended version]. *The histogram algorithm runs in $\mathcal{O}(n)$ local time and it requires a total of $\mathcal{O}(n/J)$ sample tuples.*

Proof. Lemmas 4.3.2, 4.3.3 and 4.3.5 directly imply Theorem 4.3.1. □

We next discuss why this cost is affordable. As a parallel join takes $\Omega((n+m)/J)$ communication time (m is the join output size), the histogram algorithm can afford $\mathcal{O}(n/J)$ time for collecting sample tuples and $\mathcal{O}(n)$ local processing time (which is much cheaper than the communication time).

Generalization and Discussion

We next relax some assumptions and outline how we address them to preserve all the guarantees.

A small number of output tuples. We relax the assumption $m \geq n$ from Lemma 4.3.1. If $m < n$, a sample matrix $\mathcal{M}_{\mathcal{S}}$ cell frequency can surpass m/J, breaking the Lemma bounds. We distinguish two cases. (i) If $m = \Theta(n) = cn$, where $c < 1$ is a constant, we increase n_s to preserve the bounds. More precisely, it must hold that $(n/n_s)^2 \leq m/2J$, that is, $(n/n_s)^2 \leq cn/2J$. It follows that $n_s \geq \sqrt{2nJ/c}$. Thus, n_s grows only by a constant factor of $1/\sqrt{c}$. (ii) If $c \ll 1$ ($m \ll n$), to avoid a huge increase in n_s, and thus the histogram algorithm complexity, we adjust $\mathcal{M}_{\mathcal{S}}$ such that each cell weight is below the required threshold ($w_{OPT}/2$, where w_{OPT} is the maximum region weight of the $\mathcal{M}_{\mathcal{H}}$ optimum partitioning). Namely, we divide only the row and/or column of the overweighted cell(s). Then, we reassign the affected output sample tuples to the new $\mathcal{M}_{\mathcal{S}}$ cells.

Reducing the sample matrix size n_s is an important optimization, as it decreases the histogram algorithm running time. Lemma 4.3.1 decides on n_s using a conservative assumption that $\rho_B \geq 1$ in $m = \rho_B n$, that is, $m \geq n$. (We covered the case when $m < n$ earlier in this section.) Using the actual value of $\rho_B \geq 1$ decreases n_s from $\sqrt{2nJ}$ to $\sqrt{2nJ/\rho_B}$, without loosing any guarantees. We know m and thus ρ_B from sampling the output tuples (see Section 4.4.1). This optimization requires rebuilding the sample matrix once m is known (input and output samples are collected as before). Reducing n_s is very useful when ρ_B is sufficiently bigger than 1 and when input relations are very large. We use it for B_{CB}.

Parameters. The output sample size is $s_o = \mathcal{O}(n_{sc})$. In our experiments we set $s_o = 2n_{sc}$. We compute n_{cs} by counting the candidate $\mathcal{M}_{\mathcal{S}}$ cells right after collecting a sample of input tuples.

4.7.3 Joins

The joins are defined as follows:

B_{ICD}

```
SELECT *
FROM  ORDERS O1, ORDERS O2
WHERE ABS(O1.orderkey - 10 * O2.custkey) <= 2
```

$B_{CB-\beta}$

```
SELECT *
FROM  R1, R2
WHERE ABS(R1.key - R2.key) <= β
```

BE_{OCD}

```
SELECT *
FROM  ORDERS O1, ORDERS O2
WHERE O1.custkey = O2.custkey
AND ABS(O1.ship-priority - O2.ship-priority) <= 2
AND O1.order-priority = "4-NOT SPECIFIED"
AND O2.order-priority = "1-URGENT"
AND O1.totalprice BETWEEN γ AND 360000
AND O2.totalprice BETWEEN γ AND 360000
```

For B_{CB}, we experiment with different widths of the band β: 1, 2, 3, 4, 8 and 16. Scaling out BE_{OCD} using the same γ leads to highly disturbed $output/input$ ratio ρ_{oi}. As the relative performance of different operators highly depends on ρ_{oi}, we set γ such that ρ_{oi} remains on the same order of magnitude. Namely, we set γ to 120.000, 140.000 and 160.000 for the scale factor of 80, 160 and 320, respectively.

5 Multi-way join operators: partitioning schemes and local operators

5.1 Novel join operators

We devise new join operators in Squall by wiring up state-of-the-art partitioning schemes and local join algorithms. So far, we presented novel partitioning schemes for 2-way joins. Next, we introduce multi-way joins (a multi-way join uses a single communication step, that is, it runs within a single component) in Squall. These joins can outperform the corresponding pipelines of 2-way joins as they avoid shuffling intermediate data, which can be very large [13, 144, 45]. Multi-way joins are especially beneficial when the output of intermediate stages is big compared to the size of the base relations and/or final output. Even if this is not the case, a multi-way join may outperform the corresponding pipeline of 2-way joins. In particular, an optimal query plan consisting of 2-way joins is very sensitive to the join selectivity of intermediate relations. As a query optimizer typically lacks accurate join selectivity information, it might produce a suboptimal pipeline of 2-way joins. In contract, multi-way joins are inherently resilient to inaccurate statistics [85]. In an online system, the join selectivity might vary. As we explain in Section 6.1, we could periodically adjust the join order, but the cost might be unacceptably high due to recomputing large intermediate relations. In contrast, multi-way joins inherently bring adaptivity to join selectivity variations.

We also devise a novel multi-way join partitioning scheme that further enhances performance by taking into account skew degrees of different relation attributes (see Section 5.1.2). In particular, our scheme constructs composite partitioning, consisting of different partitioning schemes according to the skew degree in different relation attributes. In addition, Squall has efficient local algorithms for online multi-way joins (DBToaster, see Section 5.1.4).

5.1.1 Applications

The need for multi-way joins arises frequently in an online scenario. Next, we mention some typical applications.

Relation	Schema
Tweet	Username, Stock symbol, Sentiment
StockTicker	Stock symbol, ΔStock price
User	Username, <User-Feature, Feature-Value> list
Result	User-Feature, Feature-Value, %Accuracy

Table 5.1: Relation and final result schemas for the Stock market query.

Mobile data analytics. A typical mobile dataset contains information about (anonymized) users, call begin and end time, and base stations used [144]. Inspired by [144, 76], we found queries that translate to multi-way joins, and that require real-time response. A query example is *Find users whose calls used multiple stations within a short period of time*. This might imply a problem (e.g. misconfiguration) in a base station. Being automatically notified by our system, the operator can immediately take appropriate actions.

Scheduling data analytics. There is a publicly available dataset with cluster monitoring data provided by Google[i]. This dataset contains information about jobs (start and end time, status, etc.), tasks (events, resource usage) and machines (assignments, attributes). We put ourselves in the shoes of a large cluster administrator, who gets notified when a potential problem arises. An interesting multi-way join query is finding machines that are not production-ready, that is, *List the machines which often fail tasks belonging to production jobs*. This is a 3-way join between jobs, tasks and machines relations. Another interesting query is *Measure the scheduling algorithm quality*. A motivation for this query is in the fact that the scheduling algorithm might perform badly for a particular (rare) event order, and this can manifest only in production. Schedulers assign jobs to machines to maximize "goodness" score [130], which includes the machine's number of preempted or failed tasks, (production) jobs distribution across the cluster, presence of application dependencies, cluster failure domains etc. For instance, it is particularly important to assign production jobs to machines with high "goodness" score. Computing the score involves joining multiple relations. We observe the scheduling algorithm quality by monitoring (in real-time) the score aggregated over jobs and machines.

Analyzing employees' performance. By continuously monitoring company's or public project repositories, we can get insights about employees' performance. For instance, a company might want to give a prize to an employee who was the result of the query *Find the employee within each team who committed the highest number of lines of code with less than 20% rewritten code for a particular language and project?* for the longest period of time. This query involves joining between Users, Commits, Projects and Teams relations.

Stock market prediction. We consider the task of stock market prediction through Twitter feeds. More precisely, we are interested in identifying classes of Twitter users who can be used as predictors of stock market activity. This can be represented as a 3-way join. Table 5.1 shows

[i] https://github.com/google/cluster-data

relations' attributes. The first relation consists of Twitter username, stock market ticker symbol and sentiments. A stream of tweets with author usernames and tagged with stock symbols is available from the Twitter's API. Sentiments can be generated by a running sentiment analysis tool over tweets. The second relation contains the stock symbols together with a percentage change in the stock's price, which can be obtained from a stock feed. These two relations are joined on the stock symbol. The third relation contains publicly available features for each user that we receive a tweet from. We join it with the other two relations on the username attribute. We say that a tweet/stock pair is accurate if the user's sentiment (positive or negative) corresponds to the stock price (up or down, respectively). Finally, the query aggregates the results, computing the average accuracy (prediction rate) for each user-feature.

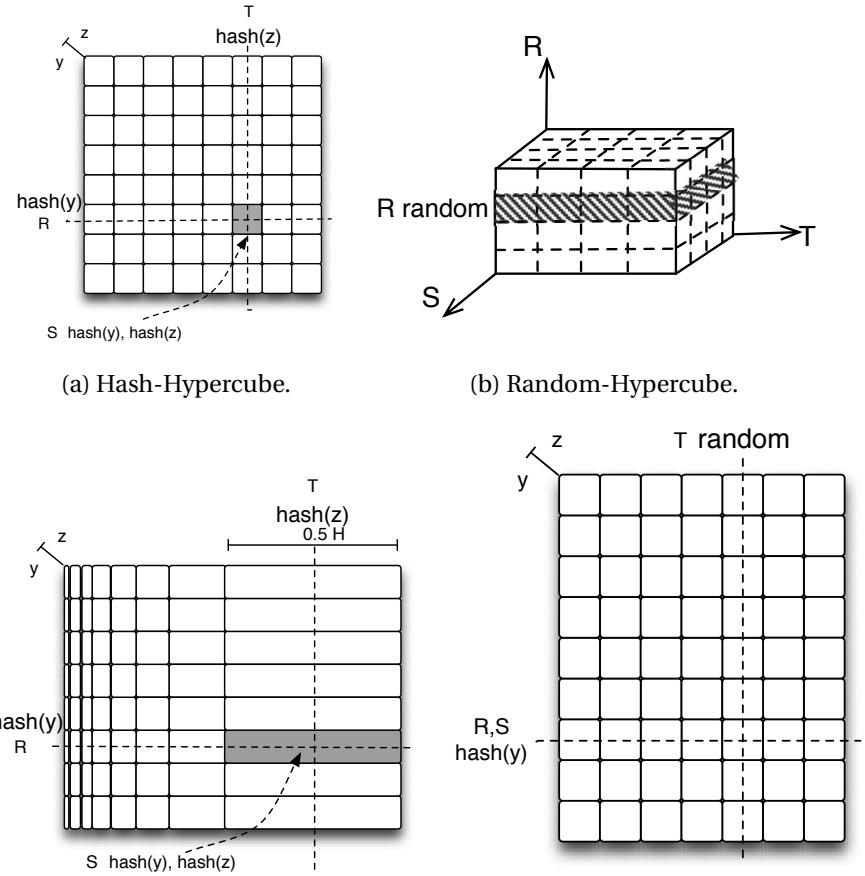
5.1.2 Partitioning schemes

Next, we describe several partitioning schemes for multi-way joins that execute a join within a single communication step, that is, within a single Squall component. We present the schemes along with their skew resilience and supported join conditions. The assumptions are that the we are dealing with a shared-nothing architecture and that initially, the data is evenly partitioned among the machines. In this section, we present the schemes briefly, along with some examples. A detailed analysis of the multi-way join schemes in Section 5.2.

Hash-Hypercube scheme [13] models the result space as a hypercube, where each axis corresponds to a join key domain. Each machine covers a unique portion of the hypercube space. Figure 5.1a illustrates this scheme for a query with a join condition $R.y = S.y$ AND $S.z = T.z$. In the further text, we refer to this query as $R(x,y) \bowtie S(y,z) \bowtie T(z,t)$. The Hash-Hypercube scheme is a generalization of hash partitioning to multi-way joins. This scheme assigns an input tuple to machines by hashing on the tuple's join keys and by replicating on the join keys from the other relations. For example, R tuples are hashed on y and replicated on z (each R tuple is replicated to a "row" of machines with coordinates $(y,z) = (hash(y), *)$). Similarly, T tuples are replicated on y and hashed on z (each T tuple is replicated to a "column" of machines with coordinates $(y,z) = (*, hash(z))$). Whereas, S tuples are partitioned using coordinates $(y,z) = (hash(y), hash(z))$. The scheme achieves correctness as each potential output tuple $t_R(x,y) \bowtie t_S(y,z) \bowtie t_T(z,t)$ is assigned to a single machine with coordinates $(hash(y), hash(z))$.

The operator's performance depends on the slowest machine, that is, the machine with the highest load (number of received input tuples). Thus, the optimization criterion is to choose the dimension sizes, such that we minimize the load per machine. In Figure 5.1a, given 64 machines and that each relation is of size H and assuming uniform distribution, the dimensions $y \times z = 8 \times 8$ minimize the load. (The dimension choosing algorithm is presented in Section 5.2.) Thus, the load of each machine L is $|R|/8 + |S|/(8 \cdot 8) + |T|/8 \approx 0.26H$. The Hash-Hypercube scheme supports skew-free multi-way equi-joins.

Random-Hypercube scheme [144]. This scheme also models the result space as a hypercube,



(a) Hash-Hypercube.

(b) Random-Hypercube.

(c) Hash-Hypercube with skew.

(d) Hybrid-Hypercube.

Figure 5.1: Partitioning schemes for $R(x,y) \bowtie S(y,z) \bowtie T(z,t)$. Uniform data (a), data-independent (b), skewed data (c, d).

but each axis corresponds to a relation, as shown in Figure 5.1b. The Random-Hypercube scheme is a generalization of the 1-Bucket scheme [106], which uses random partitioning over a matrix (2-dimensional hypercube). The Random-Hypercube scheme randomly distributes the input tuples on the axes of the originating relation, and replicates on the other axes. For example, each R tuple is replicated on a "slice" of machines (Figure 5.1b shows one such slice with diagonally engraved lines). In Figure 5.1b, given 64 machines and given that each relation is of size H, the dimensions $R \times S \times T = 4 \times 4 \times 4$ minimize the load. (The algorithm for deciding on dimensions is presented in Section 5.2.) As each machine receives $1/4$ of each relation, the load per machine is $3 \cdot H/4 = 0.75H$, where the relations are of the same size H. The Random-Hypercube scheme supports multi-way theta-joins and is skew resilient. However, it replicates tuples more than the Hash-Hypercube scheme (because it uses a 3-dimensional rather than 2-dimensional hypercube). The Random-Hypercube scheme is skew-resilient and it achieves perfect load balancing, but at the expense of the excessive tuple replication.

Our Hybrid-Hypercube scheme. Consider the same query ($R(x,y) \bowtie S(y,z) \bowtie T(z,t)$) on a

non-uniform dataset. For example, assume that y has uniform distribution and that z has zipfian distribution (the skew parameter of 2) both in S and T. The Random-Hypercube scheme performs the same independently of skew ($L = 0.75H$, as before). The Hash-Hypercube scheme with the given data distribution is shown in Figure 5.1c. Due to skew, it performs only slightly better than the Random-Hypercube (the maximum load per machine is $L = |R|/8 + |S|/(8 \cdot 2) + |T|/2 \approx 0.69H$).

Hash- and Random-Hypercube are designed and work well only for the cases when skew exists either in all the relations or in none of them. We propose the Hybrid-Hypercube, which uses hash partitioning for skew-free join keys, and random partitioning elsewhere. Random partitioning implies replication, so it is more costly than hash partitioning. That way, our scheme achieves skew resilience while minimizing tuple replication. In the case of equi-joins and skew-free attributes, the Hybrid-Hypercube produces the same partitioning as the Hash-Hypercube. Similarly, in the case of skew on all the join keys, the Hybrid-Hypercube is equivalent to the Random-Hypercube scheme. Thus, our scheme subsumes both the Hash- and Random-Hypercube schemes. Furthermore, in contrast to the Hash-Hypercube, the Hybrid-Hypercube supports non-equi joins (using random partitioning therein). For instance, our scheme works without any change if we have an inequality join condition between S and T [ii], bringing the same performance improvement compared to the Random-Hypercube as before.

The Hybrid-Hypercube scheme is illustrated in Figure 5.1d, and it works as follows. R and S tuples are hashed on y and replicated in the selected "row" of machines. We can consider $R \bowtie S$ as a (replicated) hash join. We preserve correctness as we partition R and S using the same hash function, so the corresponding partitions from these relations are on the same set of machines. Whereas, each T tuple randomly picks a "column" of machines to be replicated on. Given that there are no skew on y, $hash(y)$ from R and S simulates random distribution with respect to T. Thus, we can consider $RS \bowtie T$ as a 1-Bucket join. $RS \bowtie T$ does not indicate the order of execution, but simply the different partitioning schemes employed. We use RS rather than $R \bowtie S$ notation due to the following. As R and S use the same partitioning on y, the replication in 1-Bucket join is the same as if we had a relation of size $R + S$. We preserve correctness as follows. R and S tuples "meet" all the tuples from T, as each T tuple intersects each row on a single machine.

As a result, the maximum machine load in the Hybrid-Hypercube is $L = (|R| + |S|)/7 + |T|/9 \approx 0.36H$, which is 2.08× and 1.92× better than that of Random-Hypercube and Hash-Hypercube, respectively. It is interesting to compare these schemes with respect to total load among all the machines. The Hash-Hypercube total load is $R \cdot 8 + S + T \cdot 8 = 17H$, the total load for the Random-Hypercube is $R \cdot 16 + S \cdot 16 + T \cdot 16 = 48H$, and the one for the Hybrid-Hypercube is $R \cdot 7 + S \cdot 7 + T \cdot 9 = 23H$. Our scheme with slightly higher replication than the Hash-Hypercube (due to using random partitioning on the attributes with skew) achieves the best maximum load per machine among all the three hypercube schemes. This illustrates the tradeoff between

[ii] We only need to change the local join implementation to reflect the change in the join condition.

replication and skew resilience, which we talk about in a greater detail in Section 6.1.

5.1.3 Important special cases

Star schema typically consists of one big fact table and several small dimension tables. Usually, in a distributed setting, the fact table is partitioned and dimension tables are replicated. Interestingly, both the Hash-Hypercube and Random-Hypercube schemes comply with this partitioning. Namely, due to relative relation sizes, these schemes yield $p \times 1 \cdots \times 1$ partitioning (p is the number of machines), which implies partitioning on one dimension and replication on other dimensions. The only difference is that the Hash-Hypercube scheme partitions the fact table on join keys, while the Random-Hypercube scheme randomly partitions the fact table.

Join among multiple relations on the same key appears often in practice. An example is TPC-H [8] Q9, which joins $LINEITEM$, $PARTSUPP$ and $PART$ on $partkey$. This allows execution of a multi-way join within the same component, without any replication. Interestingly, the Hash-Hypercube scheme yields the same partitioning, as it uses the join keys as the hypercube axes.

5.1.4 Local join algorithms

Online local joins typically work as follows: a new incoming tuple for a relation is joined with the stored tuples from the other relation(s), and stored for use by future tuples [66, 58]. Existing online distributed systems enhance their local joins with indexes (hash or balanced binary tree) to improve performance. However, these joins are orders of magnitude slower than the state-of-the-art online local join, **DBToaster** [16]. The gap deepens with the increase in the number of relations in a multi-way join.

In brief, the main idea of DBToaster is to recursively maintain views for an n-way join. Instead of maintaining only the final result, DBToaster maintains all the intermediate $(n-1)$-, $(n-2)$-, $(n-3)$-, ..., and 2-way joins. For instance, given 4 relations R, S, T, V, DBToaster materializes and maintains $\binom{4}{2}$ 2-way intermediate relations ($R \bowtie S$, $R \bowtie T$, $R \bowtie V$, $S \bowtie T$, $S \bowtie V$ and $T \bowtie V$), $\binom{4}{3}$ 3-way intermediate relations ($R \bowtie S \bowtie T$, $R \bowtie S \bowtie V$, $R \bowtie T \bowtie V$ and $S \bowtie T \bowtie V$) and final result $R \bowtie S \bowtie T \bowtie V$. When a new tuple comes, DBToaster updates the intermediate relations, and produces the (delta) result by joining the incoming tuple with the corresponding $(n-1)$-way materialized join. The savings come from the fact that DBToaster does not recompute the $(n-1)$-way join for each new tuple, as it would be the case if we use indexes only on the base relations. This is why the savings grow with the increase in the number of relations n.

When parallelizing DBToaster, it is challenging to preserve correctness of the result (exactly-once semantics) as tuples (in the Incremental View Maintenance terminology, updates to relations) may arrive in different order to different machines. Existing parallel DBToaster [105] relies on a synchronous system (Spark/Spark Streaming) to circumvent the problem. As we

explain in Section 2.3, Spark Streaming is a system that performs synchronization at the end of each micro-batch, achieving latencies in the order of seconds. In contrast, Squall is an asynchronous system, as machines of the same operator make progress completely independently. Consequently, Squall achieves an order of magnitude better latencies than existing parallel version of DBToaster [105]. Furthermore, in contrast to Squall, existing parallel DBToaster [105] does not focus on skew resilience. Next, we discuss how Squall parallelizes DBToaster.

5.1.5 HyLD operator: Hypercube scheme with Local DBToaster

Squall seamlessly parallelizes the state-of-the art local join (DBToaster) by using separation of concerns. That is, Squall requires no changes in the partitioning scheme and local join when putting them together in a parallel join operator. In particular, the hypercube schemes ensure that each machine executes an independent portion of the join, so that each output tuple is produced at exactly one machine. That way, we can run a separate DBToaster instance on each machine. We denote such an operator as *Hypercube scheme with Local DBToaster* (HyLD). The HyLD operator combines network efficiency due to a hypercube scheme and CPU efficiency due to using DBToaster locally. An interested reader can find more implementation details about integration of DBToaster into Squall in Section A.1.

As we already saw, the Hybrid-Hypercube subsumes the other two hypercube schemes. Hence, it suffices to choose the right partitioning type for each dimension of the Hybrid-Hypercube. As shown in Section 5.1.2, random partitioning is expensive but skew-resilient, while hash partitioning is cheaper but prone to skew. To decide on the Hybrid-Hypercube partitioning, we need to know if join keys are skew-free or not. Note that we consider only the join keys from relations that are part of the query result. That is, we are interested in the join keys' distribution after applying selection operators over the base relations. In addition, if a relation has only a few distinct join keys, hash partitioning assigns work only to a few machines, leaving the other machines idle. In this case, we consider the relation as skewed, and use random partitioning therein.

Although DBToaster is an online local join operator, our hypercube schemes are applicable both for the offline and online scenarios. We start with the offline scenario.

Choosing among hypercube schemes: offline case. There is a threshold in attribute skew after which random partitioning brings better performance compared to hash partitioning. In offline systems, we can employ sampling and estimate the frequency of the most popular key in the dataset. Sampling incurs negligible overheads compared to the query execution time [140, 52, 107]. To find the optimal partitioning for a hypercube scheme, we run the optimization algorithm twice. In the first run, we simply compute the load after marking the attribute skewed (which enforces using random partitioning). In the second run, we run the optimization algorithm marking the attribute uniform (which opts for hash partitioning). When computing the maximum load for hash partitioning, we take into account the top key

frequency, as all the tuples with the same key go to the same machine. In particular, we estimate the maximum load per machine as $(L - L_{mf})/p + L_{mf}$, where L and L_{mf} are the load for all the keys and for the most frequent key, respectively, and p is the number of machines [iii]. Finally, we choose the partitioning (hash or random) with the smaller maximum load per machine. Alternatively, we could find out the threshold analytically. In that case, we mark the attribute as skewed or non-skewed using the information from the sample, and we run the optimization algorithm only once.

Choosing among hypercube schemes: online case. A good initial choice of a hypercube scheme saves us from future adaptations. Fortunately, in many cases, even in an online scenario, we know beforehand whether a join key is skew-free. In some cases we can infer this from the scheme. For example, an attribute with the uniqueness property (such as the primary key) cannot have skew [iv] . On the other hand, zipfian distributions are typical in many real-life datasets, including Internet packet traces, city sizes, word frequency in natural languages and advertisement clickstreams [26]. An example is dealing with chain stores, where we know ahead of time that some stores (e.g., these ones in bigger cities) sell more items than other stores. Similarly, we may know ahead of time that some products are very popular (they are sold much more frequently than other products).

5.2 Multi-way joins: General case

So far, we illustrated the Hash-Hypercube, Random-Hypercube and Hybrid-Hypercube schemes on a specific 3-way join (see Section 5.1.2). Next, we discuss the optimization algorithm for each scheme, which finds an optimal partitioning for a general join. For each scheme, the optimal partitioning produces a partitioning that minimizes the load per machine, and thus, it also minimizes the total amount of replication. We are given p machines, and the produced partitioning is a hypercube where each dimension j is of size p_j, so that $p = p_1 \cdot p_2 \cdot \cdots \cdot p_l$.

Hash-Hypercube. Given relations R_i from the query, where $i \in 1..k$, the formula for load per machine is $L = \sum_i |R_i| / \prod_{j:j \in R_i} p_j$ [13], where hypercube dimension sizes are $p_1 \times p_2 \times \cdots \times p_l$. Given the relative relation sizes (e.g., $|R_i| : |R_i| : \ldots : |R_k| = s_1 : s_2 \ldots : s_k$), the optimization algorithm chooses the dimension sizes for the Hash-Hypercube so that it minimizes the load per machine. This algorithm is known as the HyperCube algorithm [13, 29].

The formula for load L reflects the fact that the load from each relation is partitioned among dimensions that correspond to the join keys from that relation. In general, not each join key has a separate axis (equivalently, each join key corresponds to an axis, but some axes are of size 1, so we omit them from the hypercube dimensions). In contrast, previous work on the optimization algorithm [13, 28] takes as input all the attributes appearing in the query, which

[iii] We can obtain more precise estimation by using more information from the sample about data distribution, e.g., by using J most popular keys.

[iv] This holds for hash partitioning, which is a natural choice in this scenario. If we use range partitioning, we could have skew, depending on the data distribution and range bounds.

includes both join keys and the attributes from the SELECT clause (GROUP BY, aggregation attributes etc.). Indeed, we realize that using join keys is sufficient, as we will explain shortly. This observation is important as it reduces the input to the optimization algorithm, improving its performance. Using only join keys as the algorithm input also allows us to more easily reason about the optimization algorithms for the Random-Hypercube and Hybrid-Hypercube, as we will see later.

We next explain why it suffices to use only the join keys (rather than all the relation's attributes appearing in the query) as the input in the optimization algorithm. Let us relation R which has attributes x_1, x_2, ... and x_n (these are join keys), and y_1, y_2, ... and y_n (these are non-join attributes). For a fixed number of partitions p for relation R, the load per machine is the same for hypercube schemes with different dimensions from that relation. For instance, the load is the same for a hypercube scheme that uses only x_1 from R where $p = p_{x_1} = 12$, and for a scheme that has x_1, y_2 dimensions from R where $p = p_{x_1} \cdot p_{y_2} = 3 \cdot 4$. In other words, the load per machine due to relation R depends only on the number of partitions p for that relation, and not on the number of hypercube dimensions. On the other hand, only the joins keys increase the number of partitions (and reduce the load) for other relations (the ones that share the same join key). Thus, for each hypercube partitioning that contains non-join attributes, there is one which uses only join keys as dimensions, which is at least as good as the partitioning with non-join attributes. In other words, the algorithm always chooses the join keys as the hypercube dimensions, as this allows partitioning two (or more) relations with a single attribute (join key).

There are different versions of the Hash-Hypercube optimization algorithm. The original one [13] is computationally expensive as it solves a system of non-linear equations in order to find optimal dimension sizes. Beame *et al.* [28] address the efficiency problem by translating the non-linear to a linear system of equations by using some mathematical transformations. Namely, the authors express the dimension sizes in an exponential form, and then take a logarithm over the obtained mathematical expressions. The full details are outside of the scope of this thesis. For more details, we refer an interested reader to [28]. Unfortunately, as explained in [45], both works [13, 28] do not handle the case when dimension sizes (obtained from solving the equations) are not integers. For instance, if we have 7 machines in total and 3 dimensions of the same size, each dimension is of size $7^{1/3} = 1.91$. If we round down this value, we fall back to sequential execution (using only 1 machine), completely wasting the remaining 6 machines. Chu *et al.* [45] propose an algorithm that always proposes integer dimension sizes. To do so, the authors use breadth-first search to explore different configurations whose total number of machines is less or equal than the given number of machines. Then, the algorithm chooses a configuration with the smallest load per machine.

In fact, we introduce the terms Hash-Hypercube and Random-Hypercube. Furthermore, we discover and analyze the common structure between these two schemes in a principled way.

Random-Hypercube. The problem formulation is similar as before, except that the dimensions

correspond to the relations themselves, rather than to the join keys. The load per machine is equal to $\sum_i |R_i|/p_i$ [144], as each relation randomly chooses a position on its own dimension, and replicates among the other dimensions. As shown in [144], the optimal hypercube is the one that divides its dimensions into segments of equal size, that is, $|R_1|/p_1 \approx |R_2|/p_2 \approx \cdots \approx |R_k|/p_k$. In other words, in the optimal partitioning, the dimension sizes are in the same proportion as the relation sizes. For example, if we have 64 machines and R_1 is 4× bigger than R_2, the optimal partitioning is $\{R_1 \times R_2\} = \{16 \times 4\}$. This 16×4 partitioning implies the minimal load per machine and minimal communication cost among all the possible Random-Hypercube partitionings for the given proportion among the relations sizes.

We discover a technique for translating the Random-Hypercube partitioning problem to that of the Hash-Hypercube. That is, we express the join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$ as $R_1(x_1), R_2(x_2) \ldots R_k(x_k)$, where x_i are quasi-attributes that we use as the dimensions in the Hash-Hypercube optimization algorithm. As no attribute appearing in more than one relation, and each relation has exactly one attribute, the resulting partitioning scheme is the same as the one produced by the Random-Hypercube algorithm [144] for the given number of machines [v]. After we compute the dimension sizes using the Hash-Hypercube optimization algorithm, we use random rather than hash partitioning on each dimension.

Hybrid-Hypercube. To decide on dimensions and their sizes for a general multi-way join, we extend the optimization algorithm for the Hash-Hypercube. Let us first more closely look at query $R(x,y) \bowtie S(y,z) \bowtie T(z,t)$ from Section 5.1.2. The resulting Hybrid-Hypercube partitioning scheme is shown in Figure 5.1d. We obtain this partitioning by using join key renaming[vi] and by assigning each join key name to a separate hypercube dimension. In particular, given that there is skew on $S.z$ and $T.z$, we rename them to z' and z'', respectively. To address skew at join execution time, we use random partitioning on the renamed attributes z' and z''. We have to use different attribute names (z' and z''), otherwise the optimization would use the same dimension for $S.z$ and $T.z$, and as we are using random partitioning on both attributes, we would miss many result tuples. As we use separate dimensions for $S.z$ and $T.z$, and on each of them we employ random partitioning, this implies that we perform $S \bowtie T$ using the 1-Bucket scheme. On the other hand, we join R and S using hash partitioning, given that they share a common skew-free attribute y.

As we already discussed, we need to provide only join keys (rather than all the attributes from the query) as the input for the optimization algorithm. In our example, and after renaming, the input for the optimization algorithm is $R(y), S(y,z'), T(z'')$. Interestingly, the fact that renamed attributes z' and z'' use random rather than hash partitioning changes nothing in the formulas for the dimension sizes from the optimization algorithm. This is because we care only about equal distribution of tuples among the rows/columns. It is irrelevant for the formulas whether we achieve this using a hash function on a uniform dataset over by randomization. Thus, from

[v] Work [144] has an additional optimization criterion of finding the optimal operator parallelism. In our work, we assume that the number of machines is given ahead of time.

[vi] The renaming is used only in the optimization algorithm and the partitioning scheme. The local joins are unchanged.

the viewpoint of the Hash-Hypercube optimization algorithm, we can consider a renamed equi-join $R(x,y) \bowtie S(y,z') \bowtie T(z'',t)$ as an equi-join with hypercube dimensions (y,z',z'').

The Hybrid-Hypercube partitioning from Figure 5.1d has 2 rather than 3 dimensions (y,z',z''). The reason is the following. As we already discussed, an optimal partitioning includes only join keys, that is, the attributes that appear in multiple relations. Given that z' only appears in relation S, and that this relation is already partitioned by y attribute (which is a join key appearing also in R relation), the optimization algorithms sets the dimension size of z' to one, effectively removing it from the hypercube dimensions. On the other hand, although z'' is also appearing only in a single relation, it is the only attribute that partitions the relation T. Thus, attribute z'' remains in the final (y,z'') partitioning, which corresponds to our Hybrid-Hypercube from Figure 5.1d. Each tuple from R or S is hashed on y and replicated on z''. Whereas, we randomize T on z'' and replicate it on y. In other words, we perform replicated hash join between R and S, and a 1-Bucket $RS \bowtie T$ join. By doing so, the Hybrid-Hypercube saves one hypercube dimension compared to the Random-Hypercube (which directly translates to smaller amount of replication and thus better performance), while still providing for skew resilience.

Continuing this example, for certain relative relation sizes, a partitioning may become a 1-dimensional one. For instance, if T is really small compared to R and S, the optimal partitioning (with respect to the minimal load per machine) is (y), which implies broadcasting relation T. A nice property of our Hybrid-Hypercube is that it automatically handles all these cases. A user needs to provide only the relation sizes and whether each join key is skew-free or not.

Let us now consider a more complex query $R(x,y,z) \bowtie S(y,z) \bowtie T(z,t)$ in which only $T.z$ is skewed. In this case, we rename only $T.z$ to z' and use random partitioning therein. This allows us to share z attribute among R and S, lowering the amount of replication required. From the perspective of 1-Bucket join $S \bowtie T$, we simulate random distribution on $S.z$ using $hash(S.y, S.z)$, given that both $S.y$ and $S.z$ are skew-free attributes. In general, we rename attributes and create new hypercube dimensions only when necessary (in the presence of skew), allowing sharing of attributes among different relations whenever possible.

The Hybrid-Hypercube can save more than one hypercube dimension compared to the Random-Hypercube scheme. For example, if in $R(x,y) \bowtie S(y,z) \bowtie T(z,t) \bowtie U(t)$ only z has skew, the Random-Hypercube uses 4 dimensions (each corresponding to one relation), while the Hybrid-Hypercube uses only 2 dimensions (one on y attribute, and another on t). In particular, the Hybrid-Hypercube hashes R and S on attribute y to "rows" of the 2-dimensional hypercube (matrix), and T and U on t to "columns". In other words, we perform replicated hash join for $R \bowtie S$ and $T \bowtie U$, and a 1-Bucket join $RS \bowtie TU$. In order to partition the data equally using the 1-Bucket join, hashing on $S.y$ needs to produce a similar effect as random partitioning on $S.z$ (the same should hold for $T.t$ and $T.z$ attributes). This holds, as there is no skew on $S.y$ nor on $T.t$. Thus, we can apply dimensionality reduction in multiple places in

the query. In general, with the increase in the number of relations (dimensions), the potential of our hypercube scheme for saving dimensions (and reducing replication) grows. Similarly, increasing the number of relations in a pipeline of 2-way joins implies network transferring of more intermediate relations, while the corresponding hypercube scheme transfers no intermediate relations at all. On the other hand, given a fixed number of machines, increasing the dimensionality of any hypercube scheme (including ours) leads to higher replication. This is due to the fact that more dimensions have to share the same total number of machines.

Next, we analyze the Hybrid-Hypercube optimization algorithm for queries with non-equi joins. Let us consider a query $R.x = S.x$ and $S.x < T.y$. From the perspective of the optimization algorithm, we can consider this query as an equi-join $R(x), S(x), T(y)$ and dimensions (x, y) [vii]. We do not require any renaming, and we use hash partitioning for both x and y. Hash partitioning on $S.x$ allows us to reuse the same dimension for $R.x$ attribute. From the perspective of 1-Bucket join $S \bowtie T$, we simulate random distribution on $S.x$ using $hash(S.x)$, given that $S.x$ is a skew-free attribute. Similarly, we simulate random distribution on $T.y$ using $hash(T.y)$, given that $T.y$ is a skew-free attribute [viii]. That way, we perform a replicated hash join $R \bowtie S$ and an 1-Bucket join $RS \bowtie T$. In other words, the resulting partitioning scheme replicates R and S over a "row" of machines in the matrix (2-dimensional hypercube), and it replicates T over a "column" of machines.

Continuing this example, let us assume that there is skew on $T.y$. The dimensions $((x, y))$ and their sizes are the same as before. The only difference is that we need to employ random (rather than hash) partitioning on $T.y$. On the other hand, if there is skew only on $S.x$ we need to rename this attribute to x', and the optimization algorithm produces a hypercube with (x, x', y) dimensions, using hash, random and hash partitioning, respectively. In that case, attributes $R.x$ and $S.x$ correspond to different dimensions, and we employ random partitioning over the renamed attribute $S.x$ in order to handle skew.

5.3 Gathering insights about multi-way joins

In order to allow a user to easily interact with Squall, and to learn about multi-way joins and skew-resilience, we provide a graphical interface that shows the query results and various performance metrics. This work was demonstrated at VLDB 2016, where we got a very positive feedback from the demonstration attendees. We first present the overall architecture of the subsystem that collects query results and performance metrics. Then, we discuss how a user can interact with this subsystem, in order to get some insights about the data and skew-resiliency of various multi-way joins.

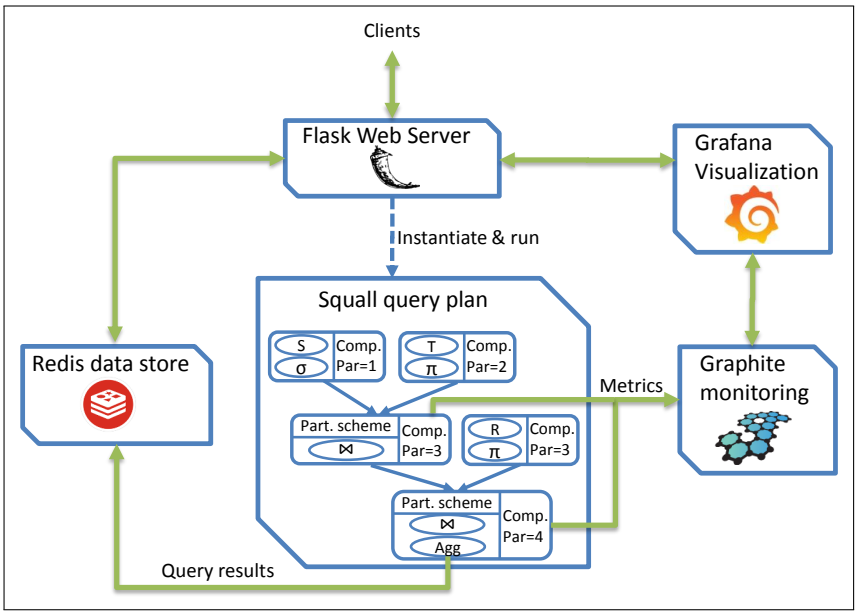


Figure 5.2: Squall's Web Interface.

5.3.1 The subsystem for collecting results and performance metrics

Figure 5.2 shows the subsystem for collecting results and performance metrics. A client communicates with a web server, and instantiates and runs a query plan. (The query plan shown in the figure contains only 2-way joins, but we use the same architecture for multi-way joins as well.) The Storm build-in web server does not show the result, and shows metrics only textually. Thus, we need to set up a separate web server and create a web page that shows both the results and performance metrics graphically. As Python code is much more concise than Java (Squall and Storm are written in Java), we opted for a Python-based web framework called Flask, which has a built-in web server. We represent the results and metrics as time series lines. This allows us not only to see the current value of a metric, but also how its changes over time, which is very convenient in a dynamic system. We discuss the interface with the user in a greater detail in Section 5.3.2.

Only the final component produces results, so we collect the results only from there. The final component sends the results to Redis, a high-performance in-memory data store. Our web application (which we set up using the Flask web server) subscribes to Redis in order to continuously get updates (the latest results in the query plan). If needed, as in the case of high update result rate and/or high parallelism of the final component, we can show a sample of the results or scale out Redis on multiple machines. We use Graphite for collecting metrics, as there are libraries that allows for easy propagation of Storm metrics to Graphite. We collect metrics from each component that contains a join. Graphite contains three modules: Carbon,

[vii] These changes are only for the optimization algorithm. The local joins are unchanged.

[viii] We could as well use random partitioning on $T.y$ in order to more closely mimic 1-Bucket partitioning for $S \bowtie T$. In that case, we do not pay any extra replication cost, as there are no other y attributes in the query.

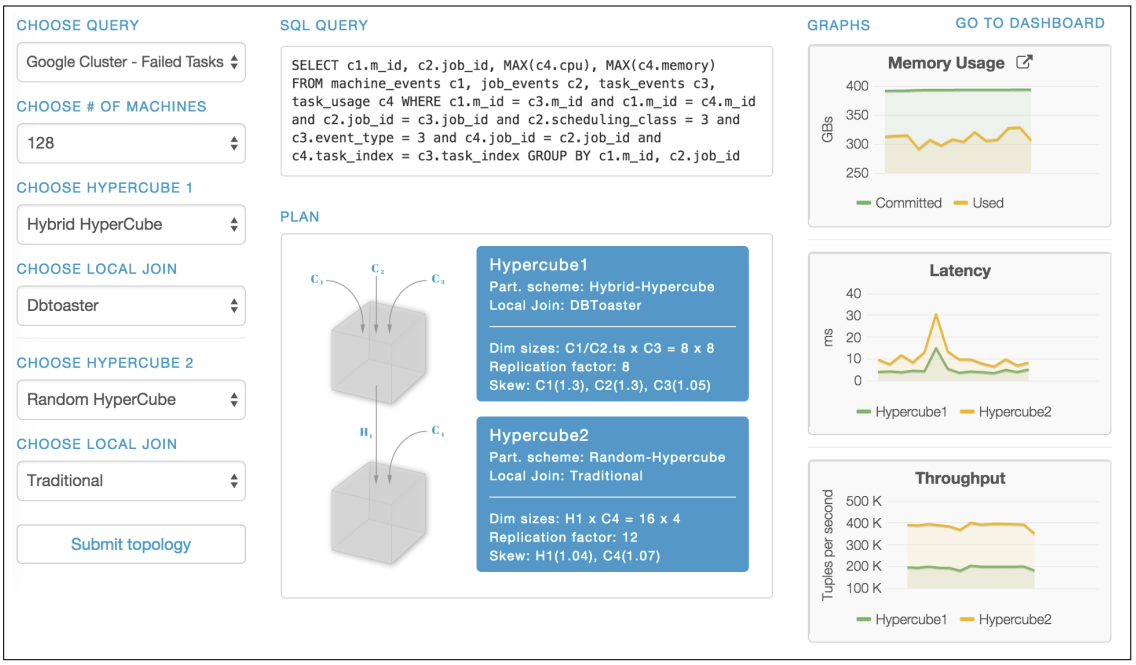


Figure 5.3: Demonstration: Running a query.

Figure 5.4: Results and query performance metrics.

Whisper and Graphite Web. Carbon collects the data and stores it in Whisper, a time-series database library. Graphite Web renders graphs from the collected data. However, we use Grafana rather than Graphite Web as a visualization tool, as Grafana offers better control offer graphs. Namely, it allows a user to specify different criteria for metrics aggregation (e.g. over machines, over time) from a graphical interface. Grafana accesses the time-series metrics data using the Graphite API. We integrate Grafana graphs into our web interface.

5.3.2 Interacting with the system

Demonstration. As shown in Figures 5.3 and 5.4, we allow attendees to specify a query and to try out different partitioning schemes (Hash-Hypercube, Random-Hypercube, Hybrid-Hypercube), local joins (traditional joins, DBToaster) and the parallelisms (number of machines). Attendees can verify scalability by changing the number of machines for a topology. With a button click, the attendees run the specified query plan on an in-house cluster with 220 hardware threads. This is illustrated with a line "Instantiate & run" in Figure 5.2. At run-time, they can continually monitor the query results, performance metrics (throughput, latency, CPU utilization and memory consumption) and operators' properties such as hypercube dimensions, replication factor and skew degree. The replication factor is the component's number of input tuples divided by the total number of tuples produced by the immediate upstream components. The replication factor is an online counterpart of the MapReduce replication rate defined in [117] as the proportion between the output and input size of the mappers in terms of number of tuples. We define skew degree as the division between the largest partition size and the average partition size.

Evaluating partitioning schemes. We allow attendees to compare hypercube schemes by monitoring the performance as a function of the operator's replication factor and skew degree. For instance, the Random-Hypercube scheme achieves perfect load-balancing (no partition skew) but it replicates tuple (as we can observe from the replication factor). For each hypercube scheme, we identify scenarios (the number of relations, their sizes and skew degrees) where it performs the best.

CPU-bound or network-bound? We aid attendees to find the bottleneck in online processing. To estimate the CPU share, we run the same query plan with different local joins (DBToaster, traditional joins). The attendees can also observe the correlation among the operator's memory consumption and throughput. To estimate the network share, we run the query plan with the same local joins but with different partitioning schemes. For instance, we replace a Hash-Hypercube with a Random-Hypercube scheme. We quantify the difference among the query plans (of the same query) using *intermediate network factor* which we define as the sum of all the component tasks' input and output divided by the sum of the query input and query output, that is, $(\sum_{comp.\ task\ t} input_t + output_t)/(query\ input + query\ output)$. The intermediate network factor represents the amount of intermediate network shuffling. Then, we compare the performance among different query plans (of the same query) as a function of this factor. The attendees can also verify on real-world queries and datasets that query plans with multi-way joins frequently outperform the ones with a pipeline of 2-way joins due to network savings.

There is an alternative way to find out if Squall query plans are CPU-bound or network-bound. We run a query plan and starting from data source reading, we add a single element (computation or network). We illustrate this process in Figure 5.5 on the example of $CUSTOMER \bowtie ORDERS$ from the TPC-H [8] dataset. For some data points, we run the query

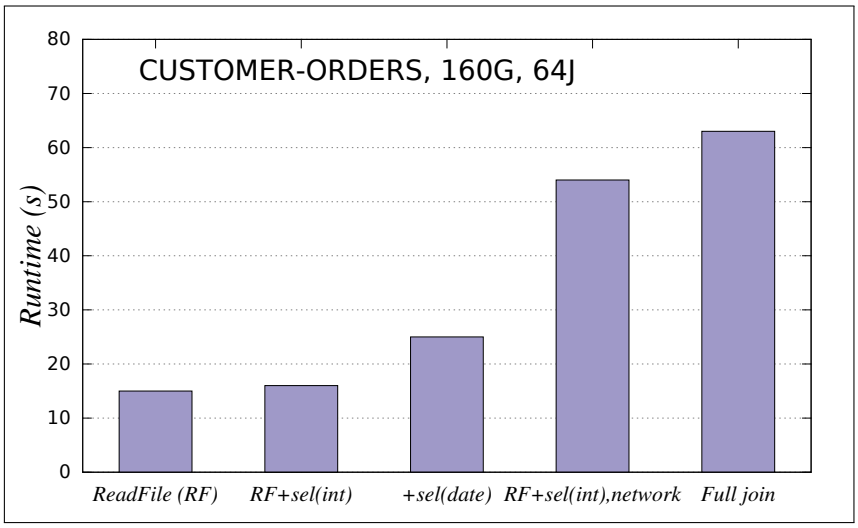


Figure 5.5: Finding bottleneck in a Squall query plan. sel stands for a no-op selection (it passes through all the tuples).

with a no-op selection (no tuples are filtered out) in order to estimate the cost of selections. The full join has no selections. From the first three bars, the cost of a selection over an integer field is only 1.6% of the entire execution. Whereas, the cost of a selection over a date field is about 16%. This is because the creation of a Date instance (from an input String) is much more expensive than the creation of an integer. From the last two bars, we extract the cost of network transferring and join computation. The network transferring takes 60% of the entire execution. Whereas, the join computation takes only 14% of the full join execution. Thus, Squall/Storm is clearly network-bound. The input throughput of the full join is 4.19 million tuples per second for the entire cluster, and 65.000 tuples per second per join task (hardware thread). Hence, our operators are fairly efficient.

5.4 Related work

Offline multi-way join schemes. The Hash-Hypercube [13] and Random-Hypercube [144] schemes, which we describe in detail in Sections 5.1.2 and 5.2, are originally proposed for offline systems. (As we show in Section 6, we can use these schemes in online systems as well, by periodically adjusting to the statistics collected so far.) Similarly, our Hybrid-Hypercube scheme is also directly applicable for offline processing. The Hybrid-Hypercube advances state-of-the-art, as in contrast to the Hash-Hypercube it supports non-equi joins and it is skew resilient, while incurring significantly smaller communication cost compared to the Random-Hypercube. The main insight of the Hybrid-Hypercube is to optimize the replication according to the join keys' skew degree and join conditions. We estimate the skew degree information from a sample from each relation.

Chu *et al.* [45] propose an operator that combines the Hash-Hypercube partitioning scheme

with a state-of-the-art offline local operator for cyclic joins. In contrast, we offer different hypercube schemes, and use state-of-the-art online local join operator for acyclic joins. Inspired by [45], in the future we plan to combine local online cyclic joins with our hypercube schemes. YSmart [86] studies partitioning schemes for subqueries consisting of both joins and aggregations. It recognizes subqueries that can be executed without any replication within a single MapReduce job.

BinHC [29] and SharesSkew [12] are partitioning schemes for multi-way joins that separate relation's tuples into heavy hitters (the join keys with high multiplicity) and light hitters (the remaining join keys). The main idea is to use some variant of hash partitioning for light hitters and random partitioning for heavy hitters. These operators reduce replication as much as possible and they may achieve smaller load per machine compared to the Hybrid-Hypercube in the offline setting. This is due to the fact that Hybrid-Hypercube always decide on partitioning according to the attribute distribution on the relation as a whole, while BinHC [29] and SharesSkew [12] partition each relation into two parts (heavy and light hitters). Thus, an optimal partitioning scheme for multi-way joins should support efficient execution of both equi-joins and non-equi joins (which our Hybrid-Hypercube does), as well as per-key partitioning for equi-joins (as BinHC [29] does). We left the design and implementation of such an operator for future work.

However, both BinHC [29] and SharesSkew [12] are restricted to equi-joins. In addition, these approaches might be suboptimal in an online scenario. In particular, they require detailed statistics about skew, that is, key frequencies. Although we can adjust the partitioning scheme according the statistics seen so far, the (relative) key frequencies can repeatedly change over time, even right after the scheme adjustment (we denote this pattern as skew fluctuations, and explain it in detail in Section 6.1). This implies frequent data migrations, which affects the performance. In contrast, the Hybrid-Hypercube requires only information about whether the relation's attribute is skew-free or not (this information is used to decide on hash or range partitioning). It does not require information about the exact degree of skew, nor about which keys are highly skewed. The skew degree on the relation as a whole typically changes less frequently than the skew degree among the particular keys, causing smaller number of migration and better performance of online Hybrid-Hypercube compared to the online counterparts of BinHC and SharesSkew.

Local online join algorithms. There is a significant body of work on local online 2-way join algorithms [137, 127, 123, 53, 98]. Symmetric hash join [137] requires that data fits in memory. Works [127, 123, 53, 98] address this issue by employing different strategies for spilling to disk. MJoin [131] generalizes XJoin [127] to (local) multi-way joins, and focuses on strategies for spilling to disk. CACQ [93] and STAIRs [49] execute multi-way joins using Eddies architecture [25], that is, they decide on per-tuple basis on an optimal join order. The main difference between DBToaster [16] that we use in Squall and these multi-way joins is as follows. First, these works [131, 93, 49] focus on equi-joins. Whereas, DBToaster also supports complex non-equi joins. Second, DBToaster materializes intermediate multi-way joins (2-way

to $(n-1)$-way joins) in order to avoid re-computation. In contrast, STAIRs only partially avoids re-computation, as it materializes intermediate tuples that results from joining of only up to 2 relations. Finally, Squall is an extensible system, as we can combine any of these local join algorithms with our partitioning schemes.

Distributed online joins. BiStream [89] and Photon [21] offer online join processing in a distributed setting. Photon [21] is designed for click-stream analytics in Google, and it supports only equi-joins. BiStream [89] is a 2-way stream join operator that partitions each input relation on a separate set of machines. It focuses on scalability and elasticity, and it supports both equi- and non-equi joins. Upon receiving an incoming tuple, BiStream always store it on exactly one machine, and produces the output by sending the tuple to all the machines that (may) contain joinable tuples from the opposite relation. BiStream uses hash partitioning (it sends an input tuple to two machines, one for storing the originating relation, and another for joining with the opposite relation) and random partitioning (an input tuple is randomly assigned to a machine of the originating relation, and sent to all the machines of the opposite relation for join processing). BiStream also proposes ContRand partitioning, which hashes an input tuple to a subgroup of machines. Within a subgroup, ContRand uses random partitioning. As BiStream always store a tuple on exactly one machine, it has smaller memory requirements than the 1-Bucket scheme [106]. However, when using random partitioning (for non-equi joins or for equi-joins with high skew), BiStream has higher communication cost that the 1-Bucket scheme [106]. We illustrate this on the following example. To simplify the analysis, we compare the two schemes assuming that the relations are of equal sizes. In that case, each relation in the BiStream scheme uses $p/2$ machines. Whereas, the 1-Bucket scheme is a $\sqrt{p} \times \sqrt{p}$ matrix. Thus, BiStream sends each tuple to $p/2$ machines, while the 1-Bucket sends a tuple only to $\sqrt{p}$ machines. In other words, the 1-Bucket scheme implies smaller communication and storage cost than the BiStream scheme.

Distributed online joins: multiple hops. We next describe the line of work that execute multi-way joins using multiple network hops. CTR scheme [68] and PSP scheme [135] optimize tuple routing, providing for adaptive join ordering. PSP [135] partitions the state among the machines according to their timestamp. CTR scheme [68] and PSP scheme [135] support both equi- and non-equi joins, These approaches attempt to address the problem of join selectivity fluctuations by adaptive join ordering. However, CTR and PSP schemes have the following drawbacks. First, these approaches do not materialize intermediate results, and suffer from recomputation. Second, the intermediate results are sent over the network and can be considerably large, causing high communication overhead, and potentially high latency for producing result tuples. In contrast, our HyLD operator solves both problems. It uses local DBToaster operator that allows reusing the previously computed intermediate results, and it requires only one network hop to produce the result tuple.

Distributed Eddies [126, 145] are based on SteMs Eddies [114], and they provide for per-tuple routing and thus, adaptive join ordering. However, Distributed Eddies [126, 145] suffer from the same drawbacks as the CTR scheme [68] and PSP scheme [135] (multiple network hops

and no intermediate relation reusing). Distributed Eddies assume window semantics, tolerate information loss and do not study intra-operator adaptations (as our Adaptive 1-Bucket scheme [58] does). Furthermore, they do not materialize intermediate results, which leads to recomputation every time a new tuple comes. For small windows, intermediate results might not be frequently reused (when window expires, its intermediate results also expire). However, reusing intermediate results is especially important for moderately-sized to large windows, and for full-history queries, which are nowadays very popular [37, 24]. Thus, we focus on large-state and full-history operators.

Distributed online joins: single hop. Next, we present the multi-way join operators that require only one network hop for producing output tuples, similarly to our hypercube schemes. ATR scheme [68] support non-equi joins and it uses range partitioning (with some overlapping) on timestamp, so it replicates tuples less than the hypercube schemes. However, ATR executes the entire window on one machine, so it might not scale for large windows and fast incoming rates. As we already discussed, online operators with large windows or full-history semantics are very popular nowadays [37, 24]. We can extend Squall with ATR partitioning schemes to support small to moderate-sized window operators.

Flux [120] is an adaptive partitioning scheme, where the number of partitions is much higher than the number of machines. This scheme supports skew but assumes that none of the partitions, which are specified in the initialization, surpasses a machine capacity. As explained in [135], this is easily violated in online scenarios. Flux is originally proposed for single-input operators, but it can support some join conditions, such as equi-joins [90]. Liu *et al.* [90, 91] provide multi-way equi-join operators using Flux, inheriting its drawbacks. Liu *et al.* [90, 91] do not consider partitioning schemes with replication, rather they focus on multi-way joins where all the relations use the same join key. This line of work offer moving operator states among the machines, as well as spilling to disk. In addition, it allows changing the join order at run-time, or even changing a pipeline of 2-way joins to a single-hop multi-way join at run-time. However, it requires blocking of input streams while migrating state. This causes long stalls for operators with large state, which is unacceptable in online systems. In contrast, our Adaptive 1-Bucket [58] is a non-blocking scheme.

5.5 Evaluation

In this section, we evaluate different hypercube schemes (Hash-Hypercube, Random-Hypercube and our Hybrid-Hypercube) for multi-way joins. We also run the corresponding pipelines of 2-way joins, where each 2-way join uses hash partitioning in the case of skew-free equi-joins, otherwise it uses the 1-Bucket partitioning. Furthermore, we compare the performance among multi-way joins with the same hypercube scheme but different local joins (DBToaster and traditional local joins). Environment (hardware setup) and programming model (Squall) is the same as in Section 4.6.1.

5.5.1 Datasets

We show the performance of our multi-way join operators both on TPC-H and on real-world datasets. The first dataset is the Hyperlink Graph of the Web from August 2012 Common Crawl Corpus [4]. In the further text, we call this dataset *WebGraph*. The WebGraph dataset has one relation with {FromUrl, ToUrl} pairs, and it is available for different domain aggregation levels. We experiment on the "Host" and "Pay-Level-Domain" aggregation levels.

Another dataset that we use is $CrawlContent$, which has crawled content from a large number of web pages [2]. We can analyze the crawled content using different tools, such as Readability test, Sentiment analysis tools etc. In the further text, $CrawlContent$ refers to a relation with the schema $\{Url, Score\}$, where $Score$ stands for the output of any text analysis tools. As the text analysis tools are out of the scope of this work, and the $Score$ is not a join key (it is used only in some aggregations), the query performance does not depend on the $Score$ values. Thus, we synthesize them.

5.5.2 Multi-way vs 2-way joins

Multi-way joins may outperform the corresponding pipeline of 2-way join, even if the corresponding pipeline is the optimal one.

3-Reachability Query. We illustrate this for a 3-step reachability query over the WebGraph dataset. The SQL of this query is shown below:

3-Reachability

```
SELECT W1.FromUrl, COUNT(*)
FROM  WebGraph as W1, WebGraph as W2, WebGraph as W3
WHERE W1.ToUrl = W2.FromUrl AND W2.ToUrl = W3.FromUrl
GROUP BY W1.FromUrl
```

This query appears frequently in practice, as it helps to understand the structure of the web. We could run the same query (with W3.ToUrl in the SELECT and GROUP BY clause) over the clickstream data, and use this information to suggest a better list of suggested hyperlinks for each website. In particular, we can propose a direct link from W1.FromUrl to W3.ToUrl, if the corresponding count aggregate value is high.

Hypercube properties. As the query contains only equi-joins, and the dataset is uniform, the Hash-Hypercube and Hybrid-Hypercube schemes produce the same partitioning. Given 36 joiners, the optimal partitioning is a 2-dimensional hypercube (matrix) $W1.ToUrl \times W2.ToUrl = 6 \times 6$, as W1, W2 and W3 are of the same size. This partitioning implies that W1 is hashed $W1.ToUrl$ and replicated on $W2.ToUrl$, W3 is hashed on $W2.ToUrl$ and replicated on $W1.ToUrl$, and W2 is hashed on both $W1.ToUrl$ and $W2.ToUrl$. Thus, the replication factor is $6+6+1 = 13$, and total network transfer due to reshuffling data is $13 \times 10.2M = 132.6M$ tuples. We run the query on 0.5% sample of the "Host" WebGraph (the full "Host" dataset has

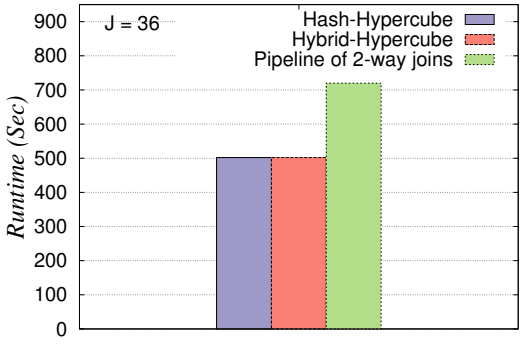


Figure 5.6: Performance for 3-reachability query. We use 36 joiner machines.

2,043 million arcs, so the sample has 10.2 million arcs), so that the pipeline of 2-way joins can also finish (otherwise, it runs out of memory due to large intermediate results). The total network transfer in the pipeline of 2-way joins is $3 \times 10.2M + 130M = 160.6M$ tuples ($130M$ is the intermediate output of the first join).

Performance results. As Figure 5.6 shows, our multi-way join outperforms the corresponding pipeline of 2-way joins by 1.43×. This is because it transfers less tuples over the network compared to the corresponding pipeline (132.6M tuples compared to 160.6M tuples). In both cases, we use DBToaster as the local join operator. The speedup comes from the fact that the intermediate results are quite large with respect to the input relations. Thus, the shuffling cost of the pipeline of 2-way joins surpasses the replication cost of a hypercube.

5.5.3 Hybrid-Hypercube versus Hash-Hypercube and Random-Hypercube

Next, we show two queries where our schemes outperforms the-state-of-the-art multi-way join partitioning schemes. All the multi-way join operators use DBToaster locally.

TPCH9-Partial Query. The first query is a subquery $Lineitem \bowtie PartSupp \bowtie Part$ from the TPC-H [8] Q9. We refer to this query as *TPCH9-Partial*. TPCH9-Partial is an example of a query where multiple relations share the same join key ($Partkey$). Wu *et al.* [138] showed that the Hash-Hypercube scheme outperforms the corresponding pipeline of 2-way joins for TPCH9-Partial on an uniform TPC-H dataset. Indeed, the Hash-Hypercube and the Hybrid-Hypercube produce the same partitioning (the hypercube is 1-dimensional with $Partkey$ as the key).

Hypercube properties. However, for a skewed TPC-H dataset, the Hybrid-Hypercube outperforms both the Hash-Hypercube and the Random-Hypercube schemes. We experiment with different configurations (J is the number of machines): $10G/8J$ and $80G/100J$ TPC-H datasets with zipfian distribution and skew factor of 2. The Hash-Hypercube scheme partitions all the relations on $Partkey$, as all the three relations have this attribute and use it as a join key.

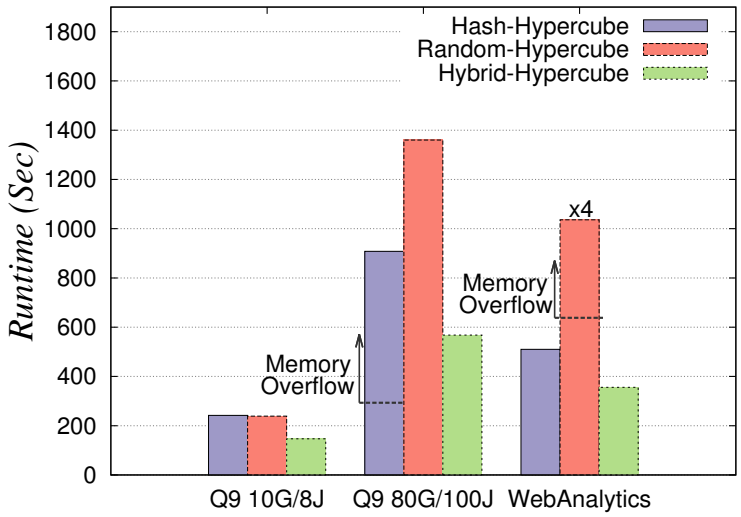


Figure 5.7: Comparison of different hypercube schemes.

The Random-Hypercube scheme uses relations as hypercube dimensions and it produces partitioning $Part \times Partsupp \times Lineitem$ with the dimensions $\{1 \times 1 \times 8\}$ for the $10G/8J$ configuration (broadcasting two smallest relations) and $\{1 \times 4 \times 25\}$ for the $80G/100J$ configuration. Due to high skew in $Partkey$ in $Lineitem$ relation, the Hybrid-Hypercube schemes uses random partitioning on $Partkey$ and hash partitioning on $Suppkey$. In particular, our Hybrid-Hypercube scheme produces $Partkey \times Suppkey$ partitioning with the dimensions $\{1 \times 8\}$ for the $10G/8J$ configuration and $\{1 \times 100\}$ for the $80G/100J$ configuration.

Performance results. Figure 5.7 shows the performance results. For query TPCH9-Partial and the $80G/100J$ configuration, the Hash-Hypercube does not complete the processing due to high memory requirements caused by high skew. However, we extrapolate its completion time using the information about the number of tuples processed before running out of memory. The Hybrid-Hypercube outperforms the Random-Hypercube by a factor of 2.39× and the Hybrid-Hypercube by 1.6×. This is due to the fact that our scheme uses hash partitioning whenever possible (on $Suppkey$) and random partitioning only when necessary due to high skew (for $PartKey$).

WebAnalytics Query. The second query that shows the advantages of our Hybrid-Hypercube scheme is over the Pay-Level-Domain WebGraph and CrawlContent datasets. It reports hyperlink paths from the WebGraph dataset that have length of two and that go through 'blogspot.com' (which has the highest in-degree in the dataset), and joins the result with the CrawlContent relation that contains URL and a web page content score. The SQL for this query is shown below:

WebAnalytics

```
SELECT W1.fromUrl, Score, COUNT(*)
FROM  WebGraph as W1, WebGraph as W2, CrawlContent as C
WHERE W1.ToUrl = 'blogspot.com' AND W2.FromUrl = 'blogspot.com'
AND W1.ToUrl = W2.FromUrl AND W1.FromUrl = C.Url
GROUP BY W1.fromUrl, Score
```

Hypercube properties. The size of WebGraph relation is 623 million arcs. After applying selections, the size of W1 and W2 is 1.03 and 3.9 million arcs, respectively. The CrawlContent relation has 43 million tuples (this is the number of distinct Urls from the Pay-Level-Domain WebGraph dataset). We compare the performance using 40 machines for each hypercube scheme. The Hash-Hypercube scheme uses a 2-dimensional hypercube with dimensions $W1.FromUrl(C.Url) \times W2.FromUrl(W1.ToUrl) = \{20 \times 2\}$. Relation $W1$ is partitioned among the machines using its FromUrl and ToUrl attributes. Relation $W2$ is hashed on $W2.FromUrl$ and replicated on $W1.FromUrl$ attribute. Whereas, relation C is hashed on $C.Url$ and replicated on $W2.FromUrl$ attribute. The Random-Hypercube scheme creates a 3-dimensional hypercube $W1 \times W2 \times C = \{1 \times 2 \times 20\}$. This schemes uses replication on all the dimensions, and relation $W1$ is replicated on all the machines. The Hybrid-Hypercube scheme creates a 2-dimensional hypercube with dimensions $W1.FromUrl(C.Url) \times W2.FromUrl = \{20 \times 2\}$. This scheme opts for random partitioning on $W2.FromUrl$ (this is optimal because WebGraph is highly skewed, as there is only one distinct value of this join key) and hash partitioning on $W1.FromUrl$ attribute (this is optimal because there is no skew on $W1.FromUrl$ and this attribute is the primary key in CrawlContent, so it is skew-free). In other words, the Hybrid-Hypercube scheme performs a replicated hash join $W1 \bowtie C$ and a 1-Bucket join $W1 - C \bowtie W2$. We use DBToaster as the local join operator for all hypercube schemes.

Performance results. Figure 5.7 shows the performance results for the WebAnalytics query. As this query takes more than an hour to execute, we show the runtime for producing the first 6.5 million output tuples (this gives us comparable running times to the ones from the TPCH9-Partial query). The Hybrid-Hypercube achieves 1.43× speedup compared to the Hash-Hypercube, and 11.64× speedup compared to the Random-Hypercube (we extrapolate its running time). This is due to the fact that, among the hypercube schemes, only our Hybrid-Hypercube scheme is able to employ different partitionings for different attributes. Furthermore, our scheme does so in an optimal manner.

Relationship between maximum load per machine and performance. To understand better the performance differences and skew resiliency of different hypercube schemes, we also extract the maximum and average load per machine in terms of number of input tuples received. Table 5.2 shows these numbers. From these numbers we can also extract skew degree, which we define in Section 5.3.2 as the division between the maximum and average load per machine. Due to the fact that the Hash-Hypercube does not address skew, it has very high maximum load compared to the average load per machine. This scheme does not complete for the TPCH9-Partial 80G configuration, and that is why we cannot obtain its maximum and average load for this configuration. In contrast, the Hybrid-Hypercube

Table 5.2: Maximum and average load per machine for different hypercube schemes. *M* stands for millions of tuples.

Query	Size	Machine Load	Hypercube type		
			Hash	Random	Hybrid
TPCH9-Partial	10G	Maximum	38.5M	15.6M	22.8M
		Average	8.5M	15.6M	8.6M
TPCH9-Partial	80G	Maximum	N/A	35M	78.9M
		Average	N/A	35M	6.3M
WebAnalytics	Pay-Level-Domain	Maximum	2.26M	N/A	2.07M
		Average	2.18M	N/A	2M

Table 5.3: Replication factor for different hypercube schemes.

Query	Size	Hypercube Replication factor		
		Hash	Random	Hybrid
TPCH9-Partial	10G	1	1.83	1.01
TPCH9-Partial	80G	N/A	6.19	1.11

addresses skew and thus it has smaller maximum load per machine than the Hash-Hypercube schemes. This explains why the Hybrid-Hypercube outperforms the Hash-Hypercube scheme, as shown in Figure 5.7. For the WebAnalytics query, it is interesting that a relatively small difference in the load (1.09×) among the Hash-Hypercube and Hybrid-Hypercube schemes leads to a considerable difference (1.43×) in the performance. This is due to the fact that this query is CPU-intensive (each incoming tuple incurs considerable computation).

The Random-Hypercube always achieves perfect load balancing due to randomization of all the input tuples. This is why the maximum and average load per machine are always the same for this scheme, but average load is rather high. In contrast, the Hybrid-Hypercube scheme replicates tuples only when necessary, and thus it has smaller average load per machine than the Random-Hypercube scheme. On the other hand, for TPCH9-Partial, the Hybrid-Hypercube has higher maximum load per machine than the Random-Hypercube, as $Suppkey$ does not have completely uniform distribution (the skew is not high enough to justify using randomization on that attribute) [ix]. Still, the Hybrid-Hypercube outperforms the Random-Hypercube scheme, as shown in Figure 5.7. Thus, we need to take into account not only the maximum load per machine but also the total communication cost (which is the average load multiplied by the number of machines), as network may be a bottleneck (i.e., network might be unable to sustain high enough throughput among all the communicating machines).

A closer look at the replication factor. Table 5.3 shows the replication factor for different hypercubes in TPCH9-Partial. We define the replication factor in Section 5.3.2 as the ratio between the total number of tuples the the component receives and the number of tuples that

[ix] When computing the Hybrid-Hypercube dimensions, for the attributes which a user marks as non-skewed, we assume uniform distribution. Thus, the computed maximum and average load per machine is the same.

the immediate upstream components (in this case, data sources) produce. A small replication factor implies small network traffic as well as small amount of local join processing. Table 5.3 illustrates that not only the Hybrid-Hypercube has lower replication factor than the Random-Hypercube (and thus better performance), but its replication factor also scales considerably better. In addition, the Hybrid-Hypercube has slightly higher replication factor than the Hash-Hypercube. However, this is exactly the reason why it is skew-resilient, and consequently, why it achieves better performance than the Hash-Hypercube scheme.

5.5.4 DBToaster versus traditional local joins

Next, we compare multi-way joins with traditional local joins versus DBToaster as the local joins.

TPC-H Queries. We run TPCH9-Partial with the $10G/8J$ configuration and TPC-H Q3 with the $10G/8J$ configuration on the TPC-H dataset with the zipfian distribution and the skew factor of 2. In all the TPC-H queries, we disregard LIMIT and ORDER BY clauses, as Squall does not support these constructs yet. The query plans with traditional joins cannot finish due to high computation cost (joiners cannot keep pace even with a minimal number of data sources), so we extrapolate their running time. The performance numbers from Figures 5.8a and 5.8b show that DBToaster brings an order of magnitude improvement compared to the traditional local joins.

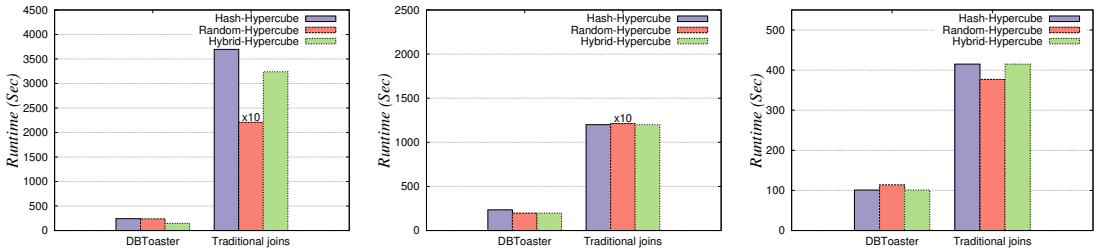
Google TaskCount Query. We also run queries over the Google scheduling dataset[x]. We presented the schema and properties of this dataset in Section 5.1.1. We provide results for a query that provides the count of failed tasks per machine id and platform over the Google scheduling dataset:

Google TaskCount

```
SELECT MACHINE_EVENTS.machineID, MACHINE_EVENTS.platform, COUNT(*)
FROM JOB_EVENTS, TASK_EVENTS, MACHINE_EVENTS
WHERE TASK_EVENTS.eventType = FAIL
AND JOB_EVENTS.jobID = TASK_EVENTS.jobID
AND MACHINE_EVENTS.machineID = TASK_EVENTS.machineID
GROUP BY MACHINE_EVENTS.machineID, MACHINE_EVENTS.platform
```

Hypercube properties. We run the Google TaskCount query using 8 machines. The Hash-Hypercube creates $machineID \times jobID = \{1 \times 8\}$ partitioning, that is, it hashes JOB_EVENTS and $TASK\_EVENTS$ and replicates the smallest relation ($MACHINE\_EVENTS$). Whereas, the Random-Hypercube produces $MACHINE\_EVENTS \times JOB\_EVENTS \times TASK\_EVENTS = \{1 \times 1 \times 8\}$ partitioning, that is, it replicates the smallest two relations ($MACHINE\_EVENTS$ and JOB_EVENTS). As the query consists of only equi-joins, and there is no significant skew, the Hybrid-Hypercube generates the same partitioning as the Hash-Hypercube scheme (recall that the Hybrid-Hypercube subsumes both the Hash-Hypercube and Random-Hypercube

[x] https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md



(a) TPCH9-Partial 10G/8J (b) TPC-H Q3 10G/8J (c) Google TaskCount 8J

Figure 5.8: Multi-way joins with different local joins (traditional vs DBToaster).

schemes). The difference between the three hypercube schemes is rather small, as the total size of $MACHINE\_EVENTS$ and JOB_EVENTS is only 14.5% of the relation $TASK\_EVENTS$ size.

Local joins. On the other hand, using different local joins makes a big difference. As Figure 5.8c shows, the hypercube schemes with DBToaster outperforms the schemes with traditional local joins by a factor of $3-4\times$.

5.5.5 Summary

Multi-way joins avoid shuffling the intermediate results, but typically have higher replication of the base relation tuples compared to the corresponding pipeline of 2-way joins. Hence, for certain queries (such as 3-Reachability), multi-way joins reduce the total communication costs. This translates to achieving better performance compared to the corresponding pipelines of 2-way joins, validating the fact that communication cost plays an important role in distributed processing. Multi-way joins are also more amenable for online processing due to their inherent adaptivity to join selectivity fluctuations. Namely, due to their hypercube structure, multi-way joins completely avoid the need for join reordering (see Section 6.1 for more information). Our Hybrid-Hypercube outperforms an existing scheme by up to an order of magnitude (see the results for query WebAnalytics in Figure 5.7). Our scheme achieves $1.6\times$ performance improvement compared to the best existing hypercube scheme (see the results for query TPCH9-Partial on $80G$ dataset using 100 machines in Figure 5.7). This is due to the fact that our scheme achieves skew resilience, while reducing replication of the tuples from base relations. The maximum and average load per machine are good performance predictors for different hypercube schemes. In general, if some of the relations are relatively small, the performance difference between the hypercube scheme drops. Finally, using DBToaster locally brings an additional speedup of up to an order of magnitude compared to the case when traditional local joins are used.

6 Adaptivity

6.1 Skew types and Adaptivity

The data distribution in an online system can change, so Squall offers some adaptivity techniques.

Skew due to hash imperfections. One may think that, in the case of uniform data distribution, hashing (both for aggregations and equi-joins) always leads to even load distribution. However, there are two situations when this is not the case. The first one happens if the number of GROUP BY/join distinct keys is smaller than the operator parallelism. It causes some machines to be completely idle. Second, uneven load distribution becomes very likely when the number of distinct keys d and the operator parallelism p are the same, or when d is a bit bigger than p. For instance, if $d = 15$ and $p = 8$, an optimal scheme will assign no more than $\lceil 15/8 \rceil = 2$ keys to each machine. However, due to imperfections of hash functions, it is very likely that some machine is assigned 3 keys, leading to 1.5× higher maximum load per machine than in an optimal case. This causes severe performance degradations. The performance gap deepens for $d = p$, as it becomes very likely that one machine is assigned 2 keys (keeping another machine completely idle), while an optimal assigns exactly 1 key per machine. The machine which is assigned two times more work becomes a bottleneck. This results in a largely suboptimal query plan in terms of resource utilization, throughput and latency, as we explained in Section 2.2.

Unfortunately, suboptimal assignments due to a small number of distinct keys d happen frequently in practice. For example, many queries from the TPC-H benchmark [8] (e.g. Q4, Q5, Q12) have final aggregations with only up to 25 distinct values. In particular, Q4, Q12 and Q5 have 5, 7 and 25 distinct values, respectively.

On the other hand, we typically know all the distinct values for attributes with a small domain (e.g. possible values for ship priorities in TPC-H are predefined). Squall uses this information to optimally assign distinct values and to achieve perfect load balancing [i]. Before the execution

[i] The optimal assignment for uniform distribution is as follows. If the number of different values is divisible by

starts, Squall creates a mapping from the predefined keys to the machines using a round robin partitioning.

Temporal skew. There is another type of skew called temporal skew, where it does not suffice for skew resilience to have the exact data distribution (even in the case of uniform distribution). Temporal skew occurs when a specific tuple arrival order causes load imbalance. In contrast to skew due to hash imperfections, temporal skew occurs only in online systems. Different partitioning schemes have different properties with respect to temporal skew. As we stated in Section 4.2, partitioning schemes are commonly classified [120] to content-sensitive schemes (e.g., joins with hash or range partitioning) and content-insensitive schemes (e.g., 1-Bucket scheme [106], which uses random partitioning). Content-sensitive schemes are prone to temporal skew. In particular, for hash partitioning, in the case of sorted tuple arrival and moderate join key frequencies, only one machine will be active at a time. This is equivalent to a sequential execution. We denote imbalance in load caused by tuple arrival order as temporal skew. Range partitioning is also prone to temporal skew. In the case of range partitioning and sorted or nearly sorted tuple arrival (e.g., a timestamp is the join key), only a few machines at a time perform some work. In the context of hypercube schemes, each scheme that uses hash partitioning on at least one dimension (with size greater than 1) is considered content-sensitive. On the other hand, content-insensitive schemes use random partitioning and they are resilient to temporal skew. Namely, these schemes perform the same independently of tuple arrival order, as the tuples are randomly distributed among the machines.

Thus, it is insufficient to capture only the data distribution. Rather, we also need to capture the temporal skew, which we can do indirectly by monitoring the machine load[ii]. To achieve good performance, we recommend using random partitioning schemes in the case of data or temporal skew (or both).

Skew fluctuations. There is an important difference in adaptivity among hash, range and random partitionings. Hash partitioning uniformly partitions the data, and thus, it always yields bad performance in the presence of skew. For range partitioning, an online operator needs to periodically adjust to the data distribution changes (e.g., when a different key becomes the one with highest frequency, or when the skew degree changes). If changes are occurring frequently, the operator spends a large amount of time on state relocations over the network. Even worse, an adversary can change the data distribution right after the system adjusts the scheme, thus causing the scheme to always be highly suboptimal. The random partitioning avoids this problem as it randomly assigns tuples to machines, essentially removing skew in the data distribution.

Join selectivity fluctuations. Next, we explain how multi-way joins bring an additional adaptivity level compared to the pipeline of 2-way joins. The join order in an optimal query plan consisting of 2-way joins is very sensitive to the join selectivity of intermediate relations. In

the number of machines, all the machines should be responsible for the same number of values. Otherwise, the number of values should not differ by more than one between any two machines.

[ii] This requires that the partitioning scheme reflects the actual data distribution.

other words, a small change in the join selectivity may cause another join order to become an optimal one. In online systems, the join selectivity for 2-way joins can vary at run-time. Furthermore, some intermediate relations may grow very large [13, 144, 45].

A possible response is adaptive join reordering [65]. In that case, we discard some intermediate relations (e.g., $R \bowtie S$) and rebuild new state for other intermediate relations (e.g., $S \bowtie T$) from scratch. This may have very adverse and hard to predict effects in an online system, including very large latencies for new incoming tuples. For this reason, existing online systems typically do not perform join reordering at run-time. Squall also do not reorder join at run-time, but it offers resilience to join selectivity fluctuations through multi-way joins.

In contrast to a pipeline of 2-way joins, a multi-way joins consists of a single join operator, so there is no need for join reordering. Furthermore, a multi-way join does not need to change which intermediate relations are materialized (e.g., $R \bowtie S$ to $S \bowtie T$ in the example above), nor to send the intermediate results over the network. Thus, in contrast to a pipeline of 2-way joins, hypercube schemes inherently bring adaptivity to the join selectivity fluctuations.

SAR principle. We introduce the SAR principle, which summarizes this section. To achieve **S**kew-resilience and **A**daptivity for more skew types in an online system, partitioning schemes need to increase the input tuple **R**eplication. Namely, for 2-way joins, hash partitioning (e.g., [66]) is prone to skew but requires no replication (hash partitioning is limited to equi-joins). Whereas, with small amount of replication, range partitioning provides resilience to Redistribution Skew (e.g., M-Bucket scheme [106]), or to both Redistribution and Join Product Skew (e.g., our equi-weight histogram scheme [132]). Unfortunately, range partitioning is prone to temporal skew and skew fluctuations. On the other hand, random partitioning (e.g., 1-Bucket scheme [106]) is resilient to data and temporal skew and skew fluctuations, but it requires a higher amount of replication compared to the one from the equi-weight histogram scheme [132]. A multi-way join is resilient to all skew types and it brings adaptivity to join selectivity variations. However, it requires higher replication than in the 1-Bucket scheme [106] due to the following. Both in 1-Bucket and multi-way joins, in order to produce the join result without requiring communication among joiner machines, a potential output tuple and all its corresponding input tuples are assigned to a single machine. Given more relations in the join, a single tuple needs to join with more tuples from other relations, effectively increasing replication. (On the other hand, pipeline of 2-way joins may incur higher total network cost compared to a multi-way join due to transferring the intermediate results over the network.)

Related work. There is a lot of work on adapting to changing input rates [123, 78, 84]. However, these works focus on a single-machine scenario, and optimizing the local join algorithm accordingly. In contrast, we introduce skew fluctuations and temporal skew, which concerns changing data distribution, and influence the choice of optimal partitioning scheme writ to their skew. Flux [120] introduces transient skew which is essentially a short-term temporal skew. The authors of [120] propose processing tuples out of order from buffers, which does not address temporal skew because all the tuples in the buffer can have a single destination.

Furthermore, Flux does not discuss the behavior of different partitioning schemes with respect to transient skew. In contrast, we reveal that only content-insensitive schemes can address temporal skew.

Regarding the SAR principle, we are the first to formalize it, the trade-off between skew-resilience and replication was known from before, both in the context of offline (e.g., 1-Bucket scheme [106]) and online processing (e.g., [65]). In contrast to the previous work, we observe the connection between adaptivity on one side, and skew-resilience and replication on the other side. We are the first to formalize the SAR principle. In addition, we provide classifications of different partitioning schemes according to their properties regarding skew-resilience (for different types of skew), adaptivity and replication.

Hypercube sizes. The optimal hypercube dimension sizes minimize replication, and thus, maximize performance. We determine the optimal sizes from the relative base relation sizes, as explained in Section 5.2. In an online system, the relative sizes may change at run-time. In that case, a hypercube scheme needs to adapt to these changes. Squall implements an adaptive 1-Bucket join operator [58], which we describe in Section 6.2.

Fault tolerance. Squall uses Storm features to achieve fault tolerance. However, we can sometimes design a better FT strategy by taking into account peculiarities of the employed partitioning schemes. In fact, if the partitioning scheme replicates tuples, a failed node can recover its state from some of its peers rather than from a disk checkpoint. For example, in Figure 5.1b, if a machine with coordinates {1, 1, 1} fails, we can recover its state from any machine {1, *, *} (for R), {*, 1, *} (for S) and {*, *, 1} (for T). This improves performance, as network accesses are several times faster than disk accesses[iii]. When RDMA is used, the performance improvements are even higher.

We can employ the same optimization even if the partitioning scheme only partially replicates the operator state. In that case, we achieve efficient fault tolerance without replicating the entire operator. Rather, we replicate only the parts of the operator state that are not already replicated by the partitioning scheme.

6.2 Adaptive 1-Bucket operator

We need data statistics to choose an optimal partitioning scheme. On the other hand, statistics may not be known ahead of time. An example is a pipeline of 2-way joins, where the input of an operator is the output of another operator. As cardinality estimation of the output is error-prone [74], we need to dynamically adjust the partitioning scheme. Similarly, data statistics is unknown in the case of reading from remote data sources [50].

Existing work on adaptive operators focus on equi-joins and partitioning on the join keys [120, 37]. In contrast, we provide adaptive 1-Bucket operator, which handles both equi-joins and

[iii] https://gist.github.com/jboner/2841832

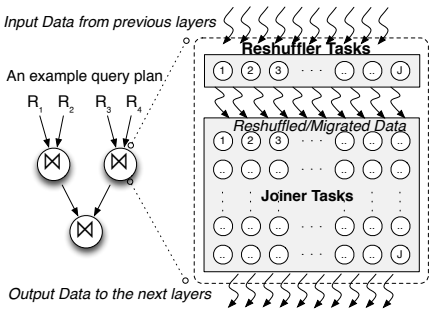


Figure 6.1: The adaptive operator structure. Each of J machines is assigned one reshuffler and one joiner task.

non-equi joins.

Our adaptive 1-Bucket operator periodically adjusts the offline 1-Bucket partitioning scheme according to the current relation sizes. Adjusting the scheme also incurs repartitioning of the operator state. Our operator collects statistics in a decentralized manner, minimizes state migration, offers a non-blocking migration algorithm, and provides optimality guarantees on data distribution and communication cost. To provide guarantees, it is crucial to choose the right moments in time (decision points) to evaluate the optimality of the current partitioning scheme, and change the scheme if necessary. We discuss each of these contributions in a separate section.

Our optimality guarantees refer to the worst-case join operator input (both in terms of data distribution and tuple arrival order). Obtaining worst-case guarantees is possible only for content-insensitive partitioning schemes for the following reason. As we explain in Section 6.1, content-sensitive schemes are either prone to data skew (e.g., hash partitioning), or to temporal skew and skew fluctuations (e.g., range partitioning). In the worst case of data distribution or tuple arrival order, only one machine is active at a time, wasting all other parallel resources. Thus, we can provide guarantees only for the content-insensitive schemes, which are resilient to these types of skew. In this section, we focus on the 1-Bucket scheme [106]). However, the 1-Bucket scheme is a 2-dimensional Random-Hypercube, and the design presented in this section generalizes to the Random-Hypercube scheme as well.

6.2.1 Operator Structure

The operator structure is shown in Figure 6.1. The operator has two types of tasks: reshufflers and joiners. Input tuples are randomly partitioned among the reshuffler tasks. Reshufflers forward input tuples to joiner tasks according to the 1-Bucket scheme. One of the reshuffler tasks is designated to collect global statistics and decide on adjusting the partitioning scheme. We denote this task as controller. Each reshuffler is aware of the currently used partitioning scheme and it replicates an input tuple among the joiner tasks in a randomly selected row or a column. A joiner tasks performs the join operation. It stores the tuples received so far, and

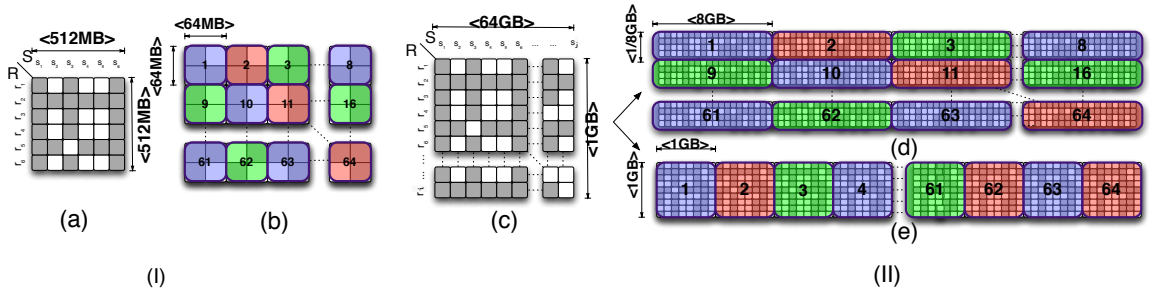


Figure 6.2: (I) Relations are of the same size (a), so 8×8 is the optimum mapping (b). (II) Both relations grow, and one relation is $64\times$ bigger than the other one (c). Using the old 8×8 mapping (d) is highly suboptimal compared to (64×1)-mapping (e).

joins new incoming tuples with the stored ones. These tasks are assigned to the J available machines such that each machine is assigned one joiner task and one reshuffler task.

6.2.2 Input-load factor

Figure 6.2 illustrates several partitioning schemes for $J = 64$ machines. For instance, Figure 6.2b) shows a (8×8)-mapping of the 1-Bucket scheme for the join matrix from Figure 6.2a). The 1-Bucket scheme is a 2-dimensional Random-Hypercube. As showed in Section 5.2, the optimal partitioning for a 2-dimensional Random-Hypercube consists of congruent rectangular regions that together cover the entire join matrix, where tuples from a region are assigned to a single machine for processing. As regions are congruent, it is sufficient to analyze the cost of a single region. Independently of the $(n \times m)$-mapping, we always divide the join matrix in J regions of the same area. The area corresponds to the number of produced output tuples, as input tuples are randomized over the join matrix. Thus, it suffices to compare the region perimeter among different $(n \times m)$-mappings of the 1-Bucket scheme. The region perimeter is $size_R \cdot |R|/n + size_S \cdot |S|/m$, where R and S are the input relations, and $size_R$ ($size_S$) is a tuple size in relation R (S). We denote the region perimeter as Input-Load Factor (ILF) [iv]. ILF represents the memory used on a machine and network traffic for sending the input tuples to that machine. Input-Load Factor also corresponds to the work for demarshaling and storing a tuple, and performing a join on a machine. Overall, by minimizing ILF, we maximize the performance. An optimal mapping is the one that minimizes ILF.

According to Section 5.2, given that the relation sizes from Figure 6.2a are equal, the (8×8)-mapping from Figure 6.2b is an optimal. This mapping has the minimal ILF among all the other possible mappings for these relations sizes. Continuing this example, assume that during the execution both relations grow such that S is $64\times$ bigger than R (Figure 6.2c). If we keep the (8×8)-mapping, the ILF is $8\frac{1}{8}$ GBs (Figure 6.2d). Whereas, the optimal mapping (obtained by using formulas from Section 5.2) is (64×1) (Figure 6.2e), and it has ILF of only

[iv] ILF is very related to maximum load per machine which is defined in the context of hypercube schemes, but the formula for ILF also contains tuple sizes.

2GBs. Consequently, it is very important to adjust the mapping during run-time.

6.2.3 Adaptivity

In this section, we discuss how to continuously adjust $(n \times m)$-mapping such that it corresponds to the optimal mapping. To that end, our adaptive operator is equipped with a control system called adaptivity loop [50], which consists of the following stages: (i) The *monitoring* stage that consists of collecting data statistics (for our operator, we need only relation sizes). (ii) The *analysis and planning* stage evaluates the optimality of the currently used scheme and explores other schemes (for our operator, we consider different $(n \times m)$-mappings). (iii) The *actuation* stage involves changing the employed mapping, which incurs data migration. In each of the stages, we have an original scientific contribution.

Monitoring statistics

Existing solutions for monitoring statistics (e.g., [120]) collect all the required data on a single machine. This machine becomes a bottleneck when the amount of collected data is large, or when the number of involved machines is big. In contrast, we only monitor the data statistics that goes through a single reshuffler (controller). As the input data is randomly partitioned among the reshufflers, the optimal mapping for the 1-Bucket scheme depends only on the relation sizes. Thus, it suffices to collect only relation sizes (rather than a histogram of the join keys).

Analysis and planning

The main challenge is when to trigger decisions, that is, when to analyze the optimality of the current mapping, and employ another mapping if necessary. If the decision points are too dense, the migration cost dominates the execution. On the other hand, if the decision points are too sparse, the operator's mapping may be highly suboptimal. In both cases, the performance is poor. We address this challenge by proposing an algorithm for triggering decisions that is constant-competitive (which implies that the ILF of the employed mapping is never worse than the ILF of the optimal one multiplied by a small constant) and that has amortized total communication cost (the cost of migrations do not dominate the communication cost of the optimal mapping in the Big O notation). Full details are in [58]. Here we provide only the intuition.

We use the fact that a large number of input tuples is necessary to significantly change the relative relation sizes. Our decision points are in exponential points in time. In particular, if one decision point occurs right after receiving $|R|$ and $|S|$ tuples, the next one is when either R reaches the size of $2|R|$, or S reaches the size of $2|S|$. We prove (Lemma 4.2 in [58]) that between two successive decision points, the optimal mapping can change only for an atomic unit. That is, if at a decision point $n \times m$ was an optimal, at the next decision point the optimal

may remain the same, or it can either be $n/2 \times 2m$ or $2n \times m/2$. Consequently, during the entire execution, the current mapping can be off the optimum mapping by at most one "unit". This provides for constant-competitiveness.

To provide for amortized communication cost, we opt for locality-aware data migration. In particular, when performing a migration, we do not reshuffle all the data stored at joiners, but we preserve as much data as possible. Let us consider the case of changing from (8×2)-mapping to (4×4)-mapping. For the former mapping, each machine keeps $|R|/8+|S|/2$ tuples. Whereas, for the latter mapping, each machine stores $|R|/4+|S|/4$ tuples. During the migration, we need to discard half of the S state on each machine, and send $R/8$ state from each machine to another machine in the operator. This cost does not dominate the total communication cost of the optimal mapping ($|R|/4+|S|/4$). The full details are in [58]. After migration, the data distribution is the same as if we used the current mapping from the beginning of the execution.

Actuation

To preserve correctness, previous work on migration algorithms (e.g., [120]) require stalling the input while performing migrations. For large-state operators, where migrations take a considerable amount of time, stalling the input incurs extremely high latencies. This is unacceptable in online scenarios.

In contrast, we process new tuples while performing migrations. In addition, we perform a migration in a completely asynchronous manner. To that end, we allow only one migration at a time and we carefully denote each tuple. There are several types of tuples: those that arrive before (τ) and those that arrive after the migration decision. As the decision about new mapping does not arrive instantly on all the reshufflers, it is possible to receive a tuple with an old mapping after the migration decision ($\triangle$). The tuples tagged with the new mapping are denoted with $\triangle'$. We ensure exactly-once semantics by producing $(\tau \cup \triangle \cup \triangle') \bowtie (\tau \cup \triangle \cup \triangle')$ for all R and S tuples. The full details are in [58].

Correctness for multiple operator groups

To simplify analysis, so far we assumed that the number of joiners is a power of two. Next, we relax that assumption. Given the number of joiners (machines) $J \in \mathbb{N}^+$, J can be uniquely decomposed into a sum of c powers of two: $J = J_1 + J_2 + \ldots + J_c$. Thus, each of c groups has J_i machines. There are $c \leq \lceil \log J \rceil$ such groups, and each group of machines runs an independent instance of Adaptive 1-Bucket. Figure 6.3 shows an example with $J = 20$ machines decomposed into 2 groups with sizes 16 and 4. Each group stores a distinct portion of the input, proportionally to its number of machines. In particular, a group i stores approximately $(J_i/J)T$ tuples, where T is the number of incoming tuples received so far. We assign incoming tuples to groups using a weighted random number generator. An incoming tuple is forwarded to all the

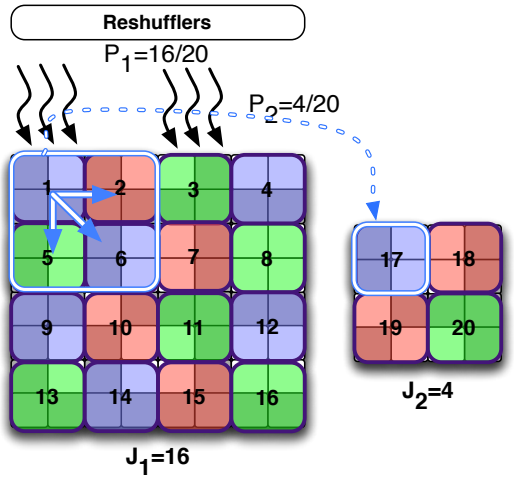


Figure 6.3: Decomposing $J = 20$ machines into independent groups of 16 and 4 machines.

groups in order to perform a join with the previously stored tuples (otherwise, some output tuples would be missed). Each incoming tuple is stored only on a single group (otherwise, the operator would produce duplicates when new tuples arrive to the system).

To operate correctly, the multi-group operator also needs to ensure the same tuple arrival order among all the groups. We illustrate this by continuing the example from above. Assume that tuples R_1 and R_2 arrive in this order on the first group, and in the reverse order (R_2, R_1) on the second group. Each tuple is stored on one group, e.g., R_2 is stored on the first group, and R_1 on the second group. In that case, always the second tuple is stored, so the result $R_1 \bowtie R_2$ is missed. Similarly, if we stored tuples differently (R_1 on the first group, and R_2 on the second group), the operator would produce duplicate results. We avoid such inconsistencies by ensuring the same order among the groups. We do so as follows.

A possible solution would be to forward all the incoming tuples through a single machine. This guarantees the same order among all the machines, given that the communication between any pair of machines is in-order (e.g., TCP is used). However, this solution limits the scalability, so we resort to an alternative solution. Given that we randomize the input tuples among the groups independently to the originating relations, the mappings of the groups will be the same (or very similar) with very high probability. This implies that a single machine in the smaller group corresponds to a block of machines in the bigger group, as shown in Figure 6.3. To ensure the same (consistent) order, we forward all the incoming tuples for a block and the corresponding machine in the smaller group through a single machine (in Figure 6.3, a machine from the block in the bigger group). In general, our protocol sends tuples serially through all the groups (at most $\log J$ of them), so tuple routing cost (and thus latency) grows at most $\log J$ times.

6.3 Towards adaptive Equi-weight histogram (EWH) scheme

As we already discussed, it is impossible to obtain worst-case guarantees for content-sensitive schemes due to temporal skew and skew fluctuations. On the other hand, general principles from our Adaptive 1-Bucket operator are reusable for content-sensitive schemes as well. In particular, we can monitor statistics in a decentralized fashion (monitoring stage), and we can design effective migration algorithms that minimize the amount of network transfers (actuation stage). The analysis and planning stage explores and triggers new partitioning schemes at certain points in time, called *decision points*. For content-sensitive operators, we can only employ a heuristic (with no guarantees) for choosing the decision points. Our adaptive EWH collects samples of tuples seen so far and at each decision point it rebuilds the equi-weight histogram scheme based on the snapshot consisting of these samples.

Regarding the operator structure, we reuse the one from the Adaptive 1-Bucket. As before, we have reshuffler and joiner tasks, and a single controller task. The only difference is that reshufflers forward tuples to joiners according to the EWH scheme, instead of the 1-Bucket scheme.

Next, we describe in detail the monitoring and actuation stages of the Adaptive EWH.

6.3.1 Monitoring Statistics

In contrast to the Adaptive 1-Bucket, it is insufficient to collect the information only about the relation sizes. Rather, we need to build the equi-weight histogram out of input and output samples. We collect input and output samples of size s_i and s_o, respectively. In a full-history online system, n is constantly growing and thus, the sample sizes need to grow as well. It is challenging to maintain a growing uniform random sample of a growing dataset. By the definition of a random sample, each input tuple has to be present in the sample with equal probability. A naive approach requires multiple passes (accessing the entire relations each time we need a sample, that is, each time we build an equi-weight histogram).

Sampling the Input Tuples. The reshufflers can produce a uniform random sample of the fixed-size input using *Bernoulli sampling with probabilistic sample size bounds* ($BERN$) [61] with the sampling rate $q_i = s_i/n$. As incoming tuples are randomly routed to reshufflers, each reshuffler operates on a uniform sample of the received tuples. Thus, it suffices for the controller to collect input statistics from a randomly chosen, *leader* reshuffler. As the reshuffler sees only $1/J$ of all the input tuples, we scale q_i to Jq_i.

We base our one-pass solution for maintaining the growing sample on the following observation: as n grows, the minimum required q_i is monotonically decreasing. This allows us to combine $BERN(q_i)$ with subsampling. Namely, the leader reshuffler periodically picks $q_i' < q_i$, informs the controller and switches to $BERN(q_i')$. After the notification, the controller subsamples its sample with $BERN(q_i'/q_i)$. It is shown in [61] that if q_i is changed based on

Algorithm 3 Assign regions to joiners.

```
1: function ASSIGN REGIONS TO JOINERS(oldRToJ, newRegions)
2:     for each region in newRegions do
3:         oldRegion = maxOverlap(region, oldRToJ)
4:         joinerId = oldRToJ.get(oldRegion)
5:         newRToJ.put(region, joinerId)
6:         oldRToJ.remove(oldRegion)
7:     return newRToJ
8: end function
```

n, rather than on the actual sample size, $BERN(q_i)$ with subsampling provides a uniform random sample of the entire population. Decreasing q_i also reduces the communication overhead.

Sampling the Output Tuples. In an online setting, joiners produce outputs as the data flows in. Thus, to build the output sample, the controller samples directly from the joiner output. Each joiner uses on its output *Bernoulli sampling with probabilistic sample size bounds* ($BERN(q_o)$) [61], with sampling rate $q_o = s_o/n = \mathcal{O}(\sqrt{J/n})$. As q_o monotonically decreases with growing n, we maintain the growing sample by combining $BERN(q_o)$ with subsampling, as before.

Combining the samples. To build an accurate sample matrix (a sample matrix which preserves the original cost distribution), the controller takes a snapshot of the input and output samples at each decision point. To do so, the controller sends a control message to the leader reshuffler, which broadcasts it to the joiners. Upon the notification, the leader and the joiners send an acknowledgement to the controller. Due to in-order delivery, the snapshot contains all the sample tuples received before the corresponding acknowledgement arrived.

Parameters. To build the sample matrix, we need to know n_{cs} and m (see §4.3.1). Interestingly, the situation is dual to the one for offline processing. Here, we easily estimate m at the controller by multiplying the sampling rate q_o with the output sample size s_o. However, it is challenging to obtain n_{cs}. By the time we build a matrix over the equi-depth histograms on the input sample, the joiners have already sent the corresponding output sample to the controller. Thus, it becomes late to change n_{cs} (q_o). To mitigate the problem, the joiners sample the output as if n_{cs} was high, and keep the sample locally. After the controller provides n_{cs}, the joiners subsample their samples and send it to the controller.

6.3.2 Actuation

In contrast to the Adaptive 1-Bucket, the joiner regions do not compose a regular grid anymore, complicating the minimization of the total migration cost. Assuming that machines can migrate state in parallel, the migration cost is dominated by the joiner with the maximum sum of the state relocated to and relocated from. We can reduce the migration cost by taking into account the current state distribution among joiners when deciding about new region-to-joiner assignment. That way, a joiner needs to relocate only a subset of its state.

A heuristic algorithm which minimizes the migration cost is shown in Algorithm 3. It works as follows. For each region from the new partitioning scheme, it finds a region from the old scheme ($maxOverlap$ function) such that the amount of retained state (overlapping input tuples) is maximized. Then, for the corresponding regions in the new and old partitioning scheme, it assigns the same joiner.

The migration algorithm requires special attention to the following. A single input tuple can be replicated in the current partitioning scheme, while in the subsequent scheme the tuple may exist only in a single region (machine). To prevent that after the migration we have two times the same tuple (which would lead to incorrect results), we introduce unique identifiers for each input tuple. This can be the primary key, a key extended with a sufficiently large random number, or a large random number (so that the probability of collisions is very small). This design allows us to assign unique identifiers at reshufflers completely independently in parallel.

On the level of system components, migration works as follows. Controller sends the new partitioning scheme and the region-joiner mapping to each joiner. According to this information, a joiner retains and sends the required parts of its local state. For quick access to transferred part of the state, given that the EWH scheme is based on range partitioning, a joiner uses a balanced binary tree index. If this type of index already exists for the join processing (e.g., in the case of a band join), it can be reused for state migration.

6.4 Evaluation for Adaptive 1-Bucket

Datasets. We run queries from the TPC-H benchmark [8]. We use the TPC-H generator [44] to generate datasets with the *Zipf* distribution. We set the degree of skew by choosing a value for the *Zipf* skew parameter z. We experiment on five different skew settings Z_0, Z_1, Z_2, Z_3, Z_4 which correspond to $z = 0$, $z = 0.25, z = 0.5, z = 0.75$ and $z = 1.0$ respectively. We create seven databases with sizes $10, 20, 40, 80, 160, 320$, and 640GB.

Queries. We run two equi-joins from the TPC-H benchmark and two synthetic band-joins. The equi-joins, E_{Q_5} and E_{Q_7}, are the most expensive join operation in queries Q_5 and Q_7. All intermediate results are materialized before processing. The two band-joins represent different workload settings. *a*) B_{CI} is a *high*-selectivity join query that corresponds to a computation-intensive workload, and *b*) B_{NCI} is a *low*-selectivity join query that represents a *non*-computation-intensive workload. The output of B_{CI} is three orders of magnitude bigger than its input size. Whereas, the output of B_{NCI} is an order of magnitude smaller than its input. The SQL for these queries are shown below. Table 4.4 summarizes all the query characteristics.

	E_{Q_5}			E_{Q_7}		
Zipf	SHJ	DYNAMIC	STATICMID	SHJ	DYNAMIC	STATICMID
$Z=0$	79	168	838*	98	192	210
$Z=1$	79	176	851*	159	183	301
$Z=2$	2742*	158	1425*	191	369	462
$Z=3$	4268*	212	2367*	5462*	334	2610*
$Z=4$	5704*	203	2849*	6385*	415	3502*

Note: [*] Overflow to disk.

Table 6.1: Runtime in secs.

B_{CI}

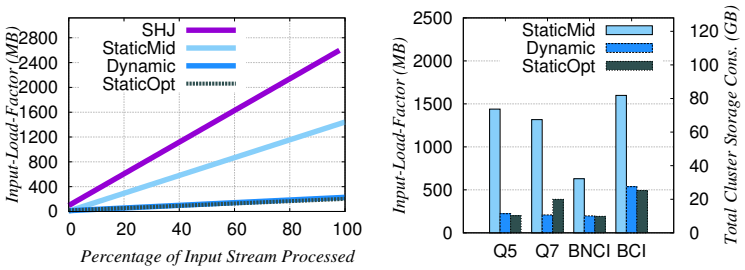
```
SELECT *
FROM  LINEITEM L1, LINEITEM L2
WHERE ABS(L1.shipdate - L2.shipdate) <= 1
AND (L1.shipmode='TRUCK' AND L2.shipmode!='TRUCK')
AND L1.Quantity>45
```

B_{NCI}

```
SELECT *
FROM  LINEITEM L1, LINEITEM L2
WHERE ABS(L1.orderkey - L2.orderkey) <= 1
AND (L1.shipmode='TRUCK' AND L2.shipinstruct='NONE')
AND L1.Quantity>48
```

Operators. We experimentally evaluate four different dataflow operators: (i) SHJ, the parallel symmetric hash-join operator described in [66]. This operator can only be used for equality join conditions and it is *content-sensitive* as it partitions data among the machines according to the join key. (ii) STATICMID, a static operator with a fixed square grid $(\sqrt{J} \times \sqrt{J})$-mapping. This scheme is efficient when both input streams have the same size and it lies in the center of the $(n \times m)$-mapping spectrum. Thus, this mapping is a best guess when no information about input stream sizes is available. (iii) STATICOPT, a static operator with an optimal mapping scheme. This requires knowledge about the input stream sizes beforehand, which is typically unknown in an online setting. (iv) DYNAMIC is our adaptive operator, which is initialized with the $(\sqrt{J} \times \sqrt{J})$-mapping scheme. Evaluation shows that our operator does not perform much worse than the STATICOPT operator, which has oracle knowledge about stream sizes (as we already discussed, the sizes are typically unknown beforehand).

Joiners perform the local join in memory, but in the case of insufficient memory, the operator starts spilling to disk. To that end, we integrate the back-end storage engine BERKELEYDB [108] in the operators. First, we experimentally verify that, in case of overflow to disk, machines suffer from long join execution times, hindering the performance. Then, for a fair comparison, we allocate sufficient memory so that all operations fit in memory (whenever possible). The heap size of each joiner machine is 2GB. Joiners use balanced binary tree indexes for band joins and hashmaps for equi-joins. We set the input data rates so that joiners are fully utilized.



(a) E_{Q_5} Input-Load Factor. (b) Final Input-Load Factor.

Figure 6.4: Input-Load Factor.

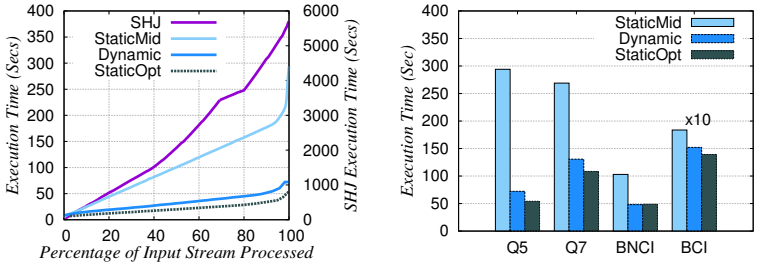
6.4.1 Skew Resilience

Table 6.1 shows experimental results for joins E_{Q_5} and E_{Q_7} with different *Zipf* skew parameter z for the 10G dataset. We compare the total execution time of our DYNAMIC and the SHJ operators using 16 machines. As expected, SHJ performs well under non-skewed settings as it does not replicate data and it evenly partitions data among machines. On the other hand, the DYNAMIC operator, partitions workload equally between joiners, but with the additional cost of tuple replication. As the skew parameter z grows (data gets more skewed), SHJ begins to suffer from poor partitioning and load imbalance among machines. This is due to the fact that most of the join work is performed on a few overwhelmed workers, while the remaining machines are mostly idle. The busy workers are assigned a large amount of input data and must overflow to disk, which severely degrades the performance. In contrast, our DYNAMIC operator is resilient to data skew and consistently partitions the data equally among machines.

6.4.2 Performance Evaluation

We compare the performance of static dataflow operators against their adaptive counterpart. We report the results for E_{Q_5} and E_{Q_7} on a Z_4 10G dataset and of B_{NCI} and B_{CI} on a uniform (Z_0) 10G dataset. We start by comparing performance using 16 machines. As shown in Table 6.1, DYNAMIC works efficiently and it consistently outperforms STATICMID. STATICMID suffers from very high values of ILF for skewed data distributions, and thus, it spills to disk, affecting the performance drastically. For a more fair comparison, we increase the number of machines to 64 so that STATICMID operator does not need to overflow to disk. In that case, STATICMID has a fixed (8×8)-mapping scheme, while the optimal mapping scheme is 1×64 for all joins. Our results show that DYNAMIC is on par with STATICOPT. This is due to the fact that DYNAMIC chooses the optimal mapping scheme very early on during the join execution. For completeness, we also evaluate E_{Q_5} and E_{Q_7} using SHJ. This operator overflows to disk due to a high degree of data skew.

Input-Load Factor. As described in Section 6.2.2, different mappings incur different values for the input-load factor. The average input-load factor for each operator shows that the the ILF has linear growth over time. We illustrate this behavior for E_{Q_5}. Figure 6.4a shows



(a) E_{Q_5} Execution Time. (b) Total Execution Time.

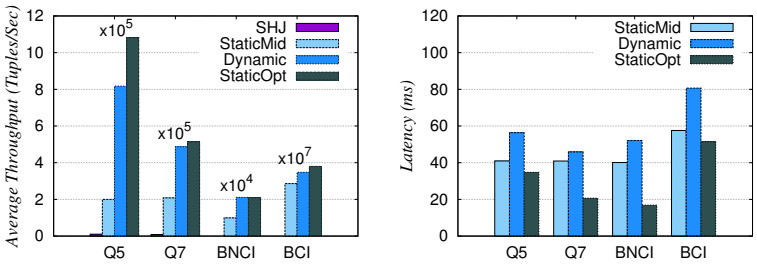
Figure 6.5: Execution Time.

the maximum size of ILF per machine as a function of the percentage of total input stream processed. SHJ, STATICMID and DYNAMIC have the growth rate of 27, 14 and 2MB per 1% input stream processed, respectively. Thus, SHJ, STATICMID suffer from high growth rates. Figure 6.4b shows graphs with the final average ILF per machine for all the queries. STATICMID consistently suffers from higher ILF values. In particular, its ILF is about 3 to 7 times bigger that of DYNAMIC. This is because the optimal mapping (1×64) (which DYNAMIC eventually chooses) lies at one end of the mapping spectrum and is far from the square grid mapping used in STATICMID. Finally, ILF in SHJ is up to 13 times that of the other operators.

In Section 6.2.2 we explain that minimizing the ILF maximizes resource utilization and performance. This is because higher ILF values leads to (i) unnecessary tuple replication among the machines in the cluster, (ii) more messages transferred over the network (potentially congesting the network), and (iii) additional overhead for processing and maintaining replicated tuples at each machine. In what follows, we evaluate the impact of ILF on operator performance.

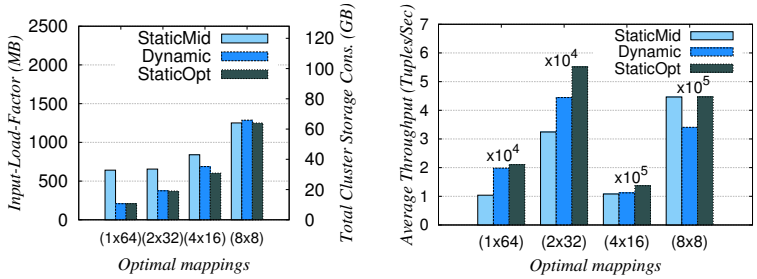
Resource Utilization. The right axis in Figure 6.4b shows the total cluster storage consumption in gigabytes. STATICMID's fixed partitioning scheme has high resource consumption (storage, network bandwidth) due to unnecessary tuple replication. Moreover, it requires 4× more machines (64) than DYNAMIC to complete its execution exclusively in main memory (for the experiments in Table 6.1, we use 16 machines). SHJ overflows to disk even when given 64 machines. DYNAMIC efficiently uses resources. This is very important in cloud environments which typically follow *pay-as-you-go* policies.

Execution Time. Figure 6.5a depicts the execution time needed to process different percentages of the input stream for query E_{Q_5}. We observe that execution time grows linearly with the percentage of input stream processed. The other join queries behave similarly. Figure 6.5b shows the total execution time for all the join queries. The join processing time directly depends on the ILF. The rigid assignment 8×8 of STATICMID yields high ILF values and thus, it consistently performs worse than DYNAMIC. However, the performance gap is smaller for computationally intensive joins (e.g., B_{CI} in Figure 6.5b). The SHJ execution time is shown at the right axis of Figure 6.5a. This operator performs two orders of magnitude worse than other



(a) Operator Throughput. (b) Tuple Latency.

Figure 6.6: Throughput and latency.



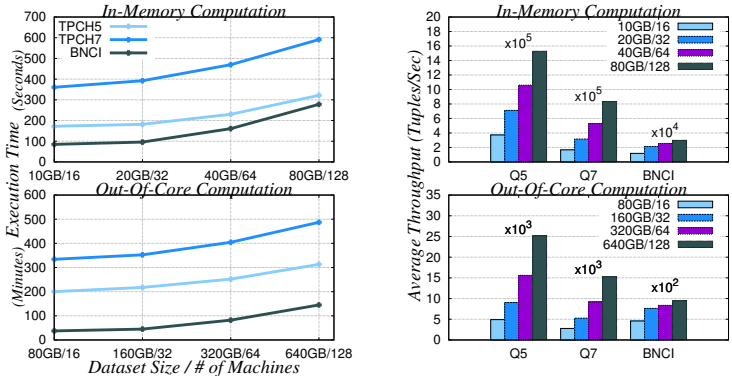
(a) Final Input-Load Factor. (b) Operator Throughput.

Figure 6.7: B_{NCI} Performance Evaluation.

operators, illustrating that poor resource utilization may cause spilling to disk, which leads to severe performance degradations. In all cases, due to the adaptivity of DYNAMIC, this operator performs very closely to STATICOPT.

Average Throughput and Latency. Figure 6.6a shows total operator throughput. The throughputs of DYNAMIC are consistently close to that of STATICOPT, and at least twice that of STATICMID. Furthermore, the throughputs of DYNAMIC are two orders of magnitude higher than that of SHJ, except for B_{CI} where the difference is slight. This validates the fact that a low ILF directly translates to high throughput, and that this effect is magnified when spilling to disk occurs. The throughput of an operator depends on the amount of join computation performed on a machine (e.g., compare B_{CI} and B_{NCI}). Fig 6.6b shows average tuple latencies. Latency is the time difference between emitting an output tuple t and the arrival of the more recent corresponding source input tuple to the operator. The figure shows that the latency is not significantly affected by the operator adaptivity. During state relocation, an extra network hop leads to an increase in the tuple latency. Overall, DYNAMIC has average tuple latency close to that of STATICMID, while it achieves much higher throughput.

Different Optimal Mappings. So far, an optimal mapping for join queries we experiment on is far from the $\sqrt{J} \times \sqrt{J}$ mapping. Next, we compare performance under different optimal mappings (see Figures 6.7a and 6.7b). To do so, we enlarge the smaller input stream. For all joins, DYNAMIC adjusts its mapping to the optimal one very early during the execution.



(a) Total Execution Time. (b) Operator Throughput.

Figure 6.8: Scalability Results.

Figure 6.7a shows that the gap in input-load factor between DYNAMIC and STATICMID drops as the optimal mapping gets closer to the grid square $(\sqrt{J} \times \sqrt{J})$-mapping scheme. Consequently, the performance gap decreases between the two operators, as Figure 6.7b shows. This validates the fact that the input-load factor has a decisive effect on performance. When $(\sqrt{J} \times \sqrt{J})$-mapping is the optimal one, all the operators (STATICOPT, STATICMID and DYNAMIC) have the same mapping, and DYNAMIC does not change its initial mapping scheme. However, DYNAMIC performs slightly worse. This is attributed to adaptivity which comes with a small, non-negligible cost.

6.4.3 Scalability Results

Next, we evaluate the scalability of DYNAMIC. In particular, we measure operator execution time and throughput as we increase both the data size and parallelism. More precisely, we evaluate weak scalability on 10GB/16 joiners, 20GB/32 joiners, and so on (see the in-memory graphs of Figures 6.8a and 6.8b). Ideally, increasing the data size/joiners configuration should result in the constant input-load factor and the output size per joiner. However, the input-load factor grows, which prevents the operator from attaining perfect scalability (doubling the data size/joiners should result in the same execution time and doubled average throughput). For example, for B_{NCI}, under the 20GB/32 configuration, the input stream sizes are 0.68M (million) and 30M tuples, respectively, and 1×32 is the optimal mapping scheme. This scheme has an ILF of $0.68M + 30M/32 = 1.61M \cdot size_{tuple}$ per joiner. However, under the 40GB/64 configuration, the input stream sizes are 1.36M and 60M, respectively, yielding a 1×64 optimal mapping scheme where an ILF is $1.36M + 60M/64 = 2.29M \cdot size_{tuple}$. In both cases, the output size per machine is the same (64K tuples). However, the ILF differs by 42% due to the tuple replication of the smaller input stream (in both cases, the smaller input stream is broadcasted, and the size of the smaller input stream increases). For the other two joins, the

ILF does not grow more than 9%. Accordingly, the execution time (Figure 6.8a) and the average throughput (Figure 6.8b) graphs show that E_{Q_5} and E_{Q_7} scale almost perfectly. For B_{NCI}, a joiner processes more input tuples as we increase the dataset size. Taking into account the increase in ILF, our operator attains good scalability.

Secondary storage. Out-of-core computation in Figures 6.8a and 6.8b shows performance under weak scalability when the operator uses secondary storage. As before, all the queries scale almost ideally, considering the growth in ILF. Thus, our operator scales with large volumes of data, and it works well for different local join algorithms. However, the performance drops by an order of magnitude compared to the *in-memory* performance results (Figure 6.8a). This validates the fact that using secondary storage is not suitable for high-performance online processing.

6.4.4 Summary

Experiments show that our adaptive operator outperforms existing *practical* static schemes in multiple performance metrics without sacrificing tuple latency. We observe that ILF has direct effect on resource utilization and performance. This justifies our optimization goal of minimizing ILF. Our operator provides efficient resource utilization (in terms of memory consumption and network bandwidth) that is up to 7 times better than that of existing non-adaptive join operators. Non-adaptivity results in higher resource requirements leading to overflows. Even when given enough resources, the adaptive operator performs up to 4 times better in terms of execution time and average throughput. We achieve adaptivity at the cost of slight increase in tuple latency (between 5 and 20ms). Experiments also validate scalability of our operator. In addition, the operator is *content-insensitive* and thus it is resilient to data skew. On the other hand, *content-sensitive* operators overflow to disk, and suffer from performance degradations of up to two orders of magnitude.

7 Conclusion

7.1 Summary of Contributions

This thesis presents Squall, a distributed online query engine. Our system automatically translates an SQL query into a DAG of Relational Algebra operators, and further to physical query plans for online processing. Skew appears in many real-world scenarios, and it is important to address it to achieve good performance. This is especially important in online systems. Thus, we focus on skew-resilient partitioning schemes for complex joins (non-equi joins, multi-way joins) and on adaptive operators.

Our partitioning schemes achieve load balancing and skew-resilience while minimizing tuple replication and the work per machine. To do so, our equi-weight histogram (EWH) partitioning scheme for monotonic 2-way joins takes into account skew and covers the spectrum of different data distributions. In contrast to previous work, our scheme ensures accurate load balancing without any prior assumptions or knowledge about the data. Rather, the EWH scheme provides an efficient parallel scheme for capturing the input and output distribution from the join to a matrix. To evenly partition the work (matrix) among the machines, the EWH scheme employs a novel multi-stage algorithm along with our join-specialized computational geometry algorithm for rectangle tiling. By doing so, we achieve up to 12× speedup and up to 5× improvement in resource utilization compared to state-of-the-art approaches.

For multi-way joins, we propose a partitioning scheme that is a composite of different partitioning schemes. We decide on the optimal scheme according to the join conditions and skew degree in different relation attributes. Our partitioning scheme performs up to an order of magnitude better than an existing partitioning scheme. We also employ state-of-the-art local join operators (DBToaster), which brings an order of magnitude performance improvement. We allow users to combine different partitioning schemes and local join operators. Such a modular system design allows practitioners to leverage the effect of various design choices on the performance. In our setup, communication costs dominate over computation costs. No matter which costs dominate, using our partitioning schemes is crucial for performance. In particular, our schemes minimize replication and network transfers per each machine (which

is important for network-bound setups). Furthermore, our partitioning schemes minimize the maximum number of tuples assigned to a machine, and thus, they minimize the computation per machine (which is important for CPU-bound setups).

Finally, we analyze the adaptive aspects of the system. We enumerate different types of skew, some of which occur only in online systems. We categorize partitioning schemes according to their resilience to different types of skew. Interestingly, an offline optimal scheme is not necessarily an online optimal scheme (e.g., in the case of temporal skew). We also reveal the SAR principle, which states that an operator with more replication is inherently more adaptive for various changes in data statistics, and it provides for skew-resilience for more skew types. Squall also offers an adaptive operator, which supports arbitrary join conditions and is capable of adjusting to changes in data statistics. This operator encapsulates data partitioning and state migration. Our adaptive operator decides when to adjust the mapping and how to perform data migration efficiently, in a non-blocking manner. The operator is highly adaptive, it has up to 4× higher throughput than its static counterpart, and it maintains low latency (on the order of tens of milliseconds). Using similar design choices, we describe how to build an adaptive counterpart of our equi-weight histogram (EWH) partitioning scheme for 2-way joins.

7.2 Future work

Squall can seamlessly parallelize various local join algorithms using any of our partitioning schemes. In addition to DBToaster, we are currently investigating efficient local algorithms for multi-way cyclic joins. A cyclic join is a join whose hypergraph, where vertices are attributes appearing in the query and hyperedges are relations from the query over the attributes, is cyclic [94]. Cyclic joins occurs frequently in practice [45]. Indeed, cyclic multi-way joins typically have the intermediate results much bigger than the final result, which makes executing within a single communication step more advantageous. Recently, worst-case optimal algorithms for local offline cyclic joins were proposed, including NPRR [104] and Leapfrog algorithm [128]. These algorithms deflect from the traditional way of executing joins which evaluates relation after a relation. Rather, they evaluate attribute by attribute, and for each attribute they process in a turn all the involved relations. Similarly to how [45] parallelizes the Leapfrog algorithm [128] using the Hash-Hypercube scheme [13], Squall could parallelize incremental version of Leapfrog [129] using any of our hypercube schemes. We also identify a need for a local join algorithm that is a hybrid between Leapfrog and DBToaster algorithms. In particular, incremental Leapfrog [129] provides certain performance guarantees regarding the worst-case join input, but only for full conjunctive queries (no projections, no aggregations). On the other hand, DBToaster is optimized for aggregation queries, and it maintains all the intermediate results, including those that might be bigger than the final result. We envision a novel hybrid join algorithm, that would use the basic structure from a worst-case optimal algorithm (e.g., incremental Leapfrog), and it would reuse previously computed intermediate results (DBToaster) whenever their sizes do not surpass the final result size.

There are many other possible extensions in Squall. One of them is supporting online aggregation. As described in Section 2.1, online aggregation continuously produces the result estimation for a static dataset according to the data processed so far. This class of processing guarantees the result quality, that is, the approximate results are within certain error bounds from the exact answer. We can use Squall to parallelize local join algorithms for online aggregation. For instance, bifocal sampling [60] is a single-machine algorithm for 2-way equi-joins that is tailored for skewed datasets. The algorithm divides the dataset into high-frequency and low-frequency keys. The design of this algorithm is reminiscent to a skew-resilient partitioning scheme for equi-joins [29]. Squall is capable of parallelizing bifocal sampling by using this scheme [29]. Similarly, for multi-way joins in the context of online aggregation, we can reuse our hypercube schemes. Ripple join [70] is a local multi-way joins for online aggregation that joins uniform random samples from multiple (N) relations. To achieve better performance, we can complement this local join with aggressive preaggregation from the Local DBToaster. Statistical tools (estimators, error bounds) from ripple joins require that we join new samples with all the previously-seen samples, and that we randomly sample from the base relations. We can satisfy both requirements in a parallel setting using the Random-Hypercube scheme [144], given that we randomize all the base relations before the processing starts. That way, the statistical tools from [70] directly apply. Thus, Squall can naturally parallelize ripple join using the Random-Hypercube scheme. However, joining samples from many relations initially tends to produce small number of output tuples, which translates to large error bounds [47]. Efficiently executing multi-way joins in the context of online aggregation is still an open research problem, even for a single-machine scenario. We anticipate that these new local joins will solicit for new partitioning schemes.

Furthermore, each query plan is currently executed in isolation (and on a separate set of machines). Frequently, these query plans share sub-plans that perform the same computation over the same input data (e.g. same selection over the same input relation). An obvious optimization is to share the common computation (and query plan operators) among multiple query plans, akin to [72, 63]. By doing so, we save some resources (machines and network traffic).

Finally, we assume flat network. This may not hold in practice, for instance, communication is faster within a rack than between racks. An important optimization is how to schedule tasks to machines considering their communication patterns (e.g., tasks that communicate a lot should be scheduled closer in the physical network) [15, 110].

A Appendix

A.1 Integrating DBToaster in Squall

DBToaster achieves good performance by recursively maintaining materialized views. From the given query, DBToaster generates these materialized views, and the code for updating them on new tuple arrivals. DBToaster provides backends in multiple languages. We use the Scala backend, because Scala runs on top of Java Virtual Machine and Squall is written in Java.

Squall invokes the DBToaster code generator while translating Squall query plan to the Storm topology. This is done on the client machine (the machine on which we submit the topology). Squall automatically invokes DBToaster, without requiring any user intervention. Then, all the required code is compiled in a single jar file, which is then submitted to the Storm cluster.

Here is an example of creating a component that runs DBToaster as the local join operator:

```
DBToasterJoinComponentBuilder dbToasterCompBuilder =
new DBToasterJoinComponentBuilder();
dbToasterCompBuilder.addRelation(relCustomer, _long);
dbToasterCompBuilder.addRelation(relOrders, _long, _long, _string, _long);
dbToasterCompBuilder.addRelation(relLineitem, _long, _double, _double);
dbToasterCompBuilder.setSQL(
"SELECT LINEITEM.f0, SUM(LINEITEM.f1 * (1 - LINEITEM.f2)) " +
"FROM CUSTOMER, ORDERS, LINEITEM " +
"WHERE CUSTOMER.f0 = ORDERS.f1 AND ORDERS.f0 = LINEITEM.f0 " +
"GROUP BY LINEITEM.f0, ORDERS.f2, ORDERS.f3");
```

In the query which is the input for the DBToaster component, relation names correspond to the upstream component names. When specifying an upstream component, we need to provide its schema. We specify fields in a relation by using the “f” letter followed by a serial order of that field in the corresponding schema.

Bibliography

[1] Amazon Architecture. http://highscalability.com/amazon-architecture.

[2] Common Crawl Corpus. http://commoncrawl.org/.

[3] Computing at CERN. https://home.cern/about/computing.

[4] Extracted Hyperlink Graph from August 2012 Common Crawl Corpus. http://webdatacommons.org/hyperlinkgraph/index.html.

[5] Graphite monitoring and visualization tool. graphite.wikidot.com/.

[6] Scalding: A scala api for cascading. https://github.com/twitter/scalding.

[7] The Apache Hadoop project. http://hadoop.apache.org.

[8] The TPC-H benchmark. http://www.tpc.org/tpch/.

[9] Trove: High Performance Collections for Java. http://trove.starlight-systems.com/.

[10] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[11] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, 1999.

[12] F. Afrati, N. Stasinopoulos, J. D. Ullman, and A. Vassilakopoulos. SharesSkew: An algorithm to handle skew for joins in mapreduce. http://arxiv.org/abs/1512.03921.

[13] F. Afrati and J. Ullman. Optimizing joins in a MapReduce environment. In *EDBT*, 2010.

[14] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[15] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 456–467. VLDB Endowment, 2004.

[16] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. In *VLDB*, 2012.

[17] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[18] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[19] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.

[20] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[21] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, 2013.

[22] Apache Flink: Scalable batch and stream data processing. https://flink.apache.org/.

[23] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford data stream management system. Technical report, Stanford InfoLab, 2004.

[24] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *VLDB*, 2004.

[25] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[26] S. Banerjee and K. Ramanathan. Collaborative filtering on skewed datasets. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 1135–1136, New York, NY, USA, 2008. ACM.

[27] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, 2010.

[28] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. http://arxiv.org/pdf/1401.1872.

[29] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, 2014.

[30] P. Berman, B. DasGupta, and S. Muthukrishnan. Slice and dice: A simple, improved approximate tiling recipe. In *SODA*, 2002.

[31] P. Berman, B. Dasgupta, S. Muthukrishnan, and S. Ramaswami. Improved approximation algorithms for rectangle tiling and packing. In *SODA*, 2001.

[32] S. Blanas, J. Patel, V. Ercegovac, J. Rao, E. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, 2010.

[33] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7(13):1441–1451, 2014.

[34] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, page 7, 2000.

[35] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *SIGMOD*, 2001.

[36] N. Bruno, Y. Kwon, and M.-C. Wu. Advanced join strategies for large-scale distributed computation. In *VLDB*, 2014.

[37] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[38] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[39] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010.

[40] B. Chandramouli, R. C. Fernandez, J. Goldstein, A. Eldawy, and A. Quamar. The Quill Distributed Analytics Library and Platform. Technical Report MSR-TR-2016-25, Microsoft Research, 2016.

[41] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[42] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD*, 1998.

[43] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, 1999.

[44] S. Chaudhuri and V. Narasayya. TPC-D data generation with skew.

[45] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, 2015.

[46] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in MapReduce. In *SIGMOD*, 2010.

[47] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[48] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[49] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[50] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1), 2007.

[51] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, 1991.

[52] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, 1992.

[53] J. Dittrich, B. Seeger, D. Taylor, and P. Widmayer. Progressive merge join: a generic and non-blocking sort-based join algorithm. In *VLDB*, 2002.

[54] T. Do and H. S. Gunawi. The case for limping-hardware tolerant clouds. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[55] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 14:1–14:14, New York, NY, USA, 2013. ACM.

[56] C. Doulkeridis and K. Nørvag. A survey of large-scale analytical query processing in mapreduce. *VLDBJ*, 23(3), 2014.

[57] P. S. Efraimidis and P. G. Spirakis. Weighted random sampling with a reservoir. *Inf. Process. Lett.*, 97(5), 2006.

[58] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. In *VLDB*, 2014.

[59] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[60] S. Ganguly, P. B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *ACM SIGMOD Record*, volume 25, pages 271–281. ACM, 1996.

[61] R. Gemulla, P. J. Haas, and W. Lehner. Non-uniformity issues and workarounds in bounded-size sampling. *VLDBJ*, 22(6), 2013.

[62] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *SOSP*, 2003.

[63] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment*, 5(6):526–537, 2012.

[64] J. D. Gibbons. *Nonparametric methods for quantitative analysis*. Holt, Rinehart and Winston, New York, 1976.

[65] A. Gounaris, E. Tsamoura, and Y. Manolopoulos. Adaptive query processing in distributed settings. *Advanced Query Processing*, 36(1), 2012.

[66] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[67] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, 1995.

[68] X. Gu, P. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *ICDE*, 2007.

[69] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. *SIGMOD*, 22(2), 1993.

[70] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

[71] L. Harada and M. Kitsuregawa. Dynamic join product skew handling for hash-joins in shared-nothing database systems. In *DASFAA*, 1995.

[72] 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*, SIGMOD '05, pages 383–394, New York, NY, USA, 2005. ACM.

[73] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *SIGMOD*, 1997.

[74] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[75] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[76] A. Iyer, L. E. Li, and I. Stoica. CellIQ: real-time cellular network analytics at scale. In *NSDI*, 2015.

[77] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. In *SIGMOD*, 2008.

[78] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

[79] O. Kennedy, Y. Ahmad, and C. Koch. Dbtoaster: Agile views for a dynamic data management system. In *CIDR*, 2011.

[80] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, 2010.

[81] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[82] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in mapreduce applications. In *Open Cirrus Summit*, 2011.

[83] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[84] R. Lawrence. Early hash join: a configurable algorithm for the efficient and early production of join results. In *VLDB*, 2005.

[85] R. Lawrence. Using slice join for efficient evaluation of multi-way joins. *Data Knowl. Eng.*, 67(1):118–139, Oct. 2008.

[86] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *Distributed Computing Systems (ICDCS)*, pages 25–36. IEEE, 2011.

[87] B. Li, Y. Diao, and P. Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015.

[88] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. J. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD Conference*, pages 985–996, 2011.

[89] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD*, 2015.

[90] B. Liu, M. Jbantova, and E. Rundensteiner. Optimizing state-intensive non-blocking queries using run-time adaptation. In *ICDE Workshop*, 2007.

[91] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005.

[92] X. Liu, N. Iftikhar, and X. Xie. Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 356–361. ACM, 2014.

[93] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[94] D. Maier. *Theory of Relational Databases*. Computer Science Press, 1983.

[95] N. Marz. STORM: Distributed and fault-tolerant realtime computation. https://github.com/nathanmarz/storm.

[96] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[97] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *VLDB*, 2010.

[98] M. Mokbel, M. Lu, and W. Aref. Hash-Merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, 2004.

[99] M. Muralikrishna and D. J. DeWitt. Equi-depth multidimensional histograms. In *SIGMOD*, 1988.

[100] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, 2013.

[101] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *ICDT*, 1999.

[102] S. Muthukrishnan and T. Suel. Approximation algorithms for array partitioning problems. *J. Algorithms*, 54(1), 2005.

[103] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*, pages 137–148. IEEE, 2015.

[104] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms:[extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[105] M. Nikolic, M. Dashti, and C. Koch. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *SIGMOD*, 2016.

[106] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, 2011.

[107] F. Olken. Random sampling from databases, 1993. PhD Thesis, UC Berkeley.

[108] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[109] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous Pig/Hadoop workflows. In *SIGMOD*, 2011.

[110] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 49–, Washington, DC, USA, 2006. IEEE Computer Society.

[111] O. Polychroniou, R. Sen, and K. A. Ross. Track join: distributed joins with minimal network traffic. In *SIGMOD*, 2014.

[112] V. Poosala and Y. E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. In *VLDB*, 1996.

[113] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.

[114] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, pages 353–364, 2003.

[115] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *VLDBJ*, 6(10), 2013.

[116] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *TODS*, 16(3), 1991.

[117] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *Proc. VLDB Endow.*, 6(4):277–288, 2013.

[118] F. R. Sayed and M. H. Khafagy. SQL TO Flink Translator. *International Journal of Computer Science Issues*, 12(1), January 2015.

[119] 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*, pages 23–34. ACM, 1979.

[120] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2002.

[121] J. Stamos and H. Young. A symmetric fragment and replicate algorithm for distributed joins. *Transactions on Parallel and Distributed Systems*, 4(12), 1993.

[122] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *VLDB*, 2001.

[123] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: producing fast join results on streams through rate-based optimization. In *SIGMOD*, 2005.

[124] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.

[125] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a MapReduce framework. In *VLDB*, 2009.

[126] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.

[127] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), 2000.

[128] T. L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[129] T. L. Veldhuizen. Incremental maintenance for leapfrog triejoin. *CoRR*, abs/1303.5313, 2013.

[130] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems*. ACM, 2015.

[131] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.

[132] A. Vitorovic, M. Elseidy, and C. Koch. Load balancing and skew resilience for parallel joins. In *ICDE*, 2016.

[133] W. Vogels. Eventually consistent. *ACM Queue*, 2008.

[134] C. Walton, A. Dale, and R. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *VLDB*, 1991.

[135] S. Wang and E. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. In *EDBT*, 2009.

[136] Y. Wang. Relations between two common types of rectangular tilings. *Int. J. Comput. Geometry Appl.*, 19, 2009.

[137] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Parallel and Distributed Information Systems*, 1991.

[138] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.

[139] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.

[140] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, 2008.

[141] W. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.

[142] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[143] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.

[144] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using MapReduce. *VLDBJ*, 5(11), 2012.

[145] Y. Zhou, B. Ooi, and K. Tan. Dynamic load management for distributed continuous query systems. In *ICDE*, 2005.

[146] G. K. Zipf. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books, 2016.

Aleksandar Vitorović

Av. de Préfaully 25A | Chavannes-pres-Renens, VD | aleksandar.vitorovic@epfl.ch | +41 (0)78 610-81-39

STRENGTHS

- I love solving hard problems in Big Data/Databases that bring significant savings to company. At Microsoft, I introduced changes in a production distributed system, and achieved up to 1.75X savings in resources needed.
- During my PhD, I built Squall, a distributed query engine that is up to 15X faster than existing work. Squall attracted attention from industry (Huawei).
- Squall is an open-source project, where a total of 24 people contributed. I led it through the design, development and maintenance stages.

EDUCATION

PhD in Computer Science with Prof. Christoph Koch | EPFL
September 2010 - | Lausanne, VD

M. Sc. in CS | SCHOOL OF ELECTRICAL ENGINEERING
May 2009 – July 2010 | University of Belgrade, Serbia

B. Sc. in CS | SCHOOL OF ELECTRICAL ENGINEERING
October 2004 – February 2009 | University of Belgrade, Serbia

- 3-months exchange at Texas A&M University, United States

PROFESSIONAL EXPERIENCE/ INTERNSHIPS

PhD project in Big Data/Databases | EPFL
September 2010 - | Lausanne, VD

- Squall (50K+ lines of Java code), a distributed low-latency query engine built on top of Apache Storm. Main features:
  - Efficient and adaptive **skew-resilient joins** (up to 15X speedup)
  - **Modularity**: combine best data partitioning and local operators
  - **Scalability** (I run Squall on a cluster of 220 hardware threads)
  - A resource-aware **query optimizer**
- Presented Squall to the managers of Huawei's StreamSMART system in an Invited visit to Huawei Research Lab, Hong Kong (September 2014.).

INTERN | MICROSOFT RESEARCH
May 2015 – August 2015 | Redmond, WA, USA

- The company needed to run more production queries on an existing cluster (small errors acceptable). Worked in C++ on Approximate Query Processing.
- Participated in the design of sampler operators and in the modification of a production query optimizer (result: 2X fewer resources needed).
- Extended the Microsoft SCOPE engine to natively support samplers (result: additional savings in resources up to 1.75X).

INTERN | TEXAS A&M UNIVERSITY
September 2008 – December 2008 | College Station, TX, USA

- Advised by Prof. Lawrence Rauchwerger.
- Parallelization of SPEC CPU2006 C++ code using OpenMP.
- SPEC is hard to parallelize; I found patterns that allow for up to 6x speedup.
- I used this work as my Master thesis.

DEVELOPER-CONTRACTOR | INSTITUTE MIHAILO PUPIN
February 2009 – July 2010 (hourly payed, average 6h/day) | Belgrade, Serbia

- Renewable Energy Sourcing Decisions and Control: EU-FP7 project.
- Built a prototype web simulator, and coupled Java JSF and MATLAB.

LUMP-SUM CONTRACTOR | SERBIAN OBJECT LABORATORIES
July 2006 – February 2007 | Belgrade, Serbia

- Transforming UML models into Java code: SOLoist. Designed and implemented UML model checker, and tested the Java code generator.

PUBLICATIONS

- A. Vitorović, M. Elseidy, K. Gyliev, K. Vu Minh, D. Espino, M. Dashti, Y. Klonatos and C. Koch. *Squall: Scalable Real-time Analytics*. VLDB Demo 2016. Full version of the paper is available as a Technical Report.
- A. Vitorović, M. Elseidy and C. Koch. *Load Balancing and Skew Resilience for Parallel Joins.* ICDE 2016. Full version of the paper is available as a Technical Report.
- S. Kandula, A. Shanbhag, A. Vitorović, M. Olma, R. Grandl, S. Chaudhuri and B. Ding. Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters. SIGMOD 2016, patented.
- M. Elseidy, A. Elguindy, A. Vitorović and C. Koch. *Scalable and Adaptive Online Joins.* VLDB 2014. Full version of the paper is available as a Technical Report.
- A. Vitorović, M. Tomašević and V. Milutinović. *Manual Parallelization Versus State-of-the-Art Parallelization Techniques: The SPEC CPU2006 as a Case Study.* Advances in Computers, Vol. 92., 2014.
- D. Paunović, A. Vitorović, and M. Batić. *Web enabling of the Matlab / Simulink models*. YUINFO 2011.

TEACHING ASSISTANT

- Big Data: Spring 2014, 2015, graduate-level, 120 students.
- Advanced Databases: Spring 2011, 2012, 2013, graduate-level, 100 students.
- Principles of Computer Systems: Fall 2012, graduate-level, 25 students.
- Linear Algebra: Fall 2013, 2014, undergraduate-level, 300 students.
- Java Programming: Fall 2015, undergraduate-level, 150 students.

UNIVERSITY PROJECTS (during PhD, M. Sc. and B. Sc.)

- *HadoopGunzip*: Gunzipping binary files in Hadoop. This was a homework in the Big Data class, and I was the responsible TA. I had to implement it, and the solution required changes in the Hadoop source code. I proposed 3 solutions (Map-only Hadoop, MapReduce Hadoop and Map-only Hadoop Streaming Job) with different tradeoffs in performance and maintainability.
- *ParImage*: A parallel image processor in Java w/ a server and worker nodes.
- *SVMeter*: A C++ Linux application that aggregates network traffic over time.
- *SmallOS*: A small operating system in C++ w/ timesharing and multithreading. It can also connect to a driver that simulates disk.
- *CitySearch*: Finding shortest paths in a city using A* and branch and bound search algorithms.
- *MemEff*: Given a distribution of programs' memory requirements, it yields the average memory utilization.

TECHNICAL SKILLS

- **Programming:** Java (2 years industry + 5 years academia) w/ JSP and JSF, C, C++ w/ OpenMP, C#, Python, Thrift, OCaml, Visual Basic, SQL, MATLAB
- **Parallel frameworks:** Apache Storm, Hadoop, Heron, Microsoft SCOPE
- **Relational Databases:** Microsoft SQL Sever, MySQL
- **Operating systems:** Linux and Solaris (including bash), Windows

HONORS & AWARDS

- Best 4-th year Student GPA Award (2008.)
- Eurobank EFG Award for 100 Best Students in Serbia (2007.)
- Scholarship from the Serbian Government (2004.-2010.)
- Scholarship from Foundation for young talents - Dositeja, Serbia (2010.-2014.)

LANGUAGES

- **English** (C1), **French** (B1/B2 according to DIALANG), **Serbian** (native)

PERSONAL INFO

- 30 years

REFERENCES

- Available upon request.