

Efficient Online Processing for Advanced Analytics

THÈSE N° 7731 (2017)

PRÉSENTÉE LE 26 OCTOBRE 2017

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE THÉORIE ET APPLICATIONS D'ANALYSE DE DONNÉES

PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Mohamed Elsayed Mohamed Ahmed EL SEIDY

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury

Prof. C. Koch, directeur de thèse

Prof. M. Püschel, rapporteur

Dr C. Curino, rapporteur

Prof. K. Aberer, rapporteur



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2017

*Nothing in this world can take the place of persistence. Talent will not;
nothing is more common than unsuccessful men with talent. Genius will not;
unrewarded genius is almost a proverb. Education will not;
the world is full of educated derelicts.
Persistence and determination alone are omnipotent.
The slogan Press On! has solved and always will solve the problems of the human race.*
— Calvin Coolidge

To my long gone father Elsayed who has left a young seed behind and
to my mother Soad who continued to water the seed till it thrived.



Acknowledgements

This Ph.D. dissertation would not have been possible without the support of my advisor, mentors, colleagues, family, and friends. Most importantly, I would like to thank my advisor Prof. Christoph Koch for mentoring and guiding me through the Ph.D journey. I thank him for devoting all his energy, time, and experience to pave a successful path ahead of me. He has inspired me and taught me the principles of critical thinking and abstract reasoning.

I am also thankful to my thesis committee members, Prof. Willy Zwaenepoel, Prof. Karl Aberer, Dr. Carlo Curino, and Prof. Markus Puschel, for giving their time and energy to evaluate my thesis and to provide insightful feedback. I am indebted to Prof. Willy for continuously supporting and believing in me since the beginning of the journey. Prof. Karl has taught me the first principles of distributed processing through his class. I am also grateful to Dr. Carlo for being a supportive, friendly, and inspiring mentor during my internship at Microsoft Research and for encouraging me to explore new research directions. I would also like to thank Prof. Markus for giving me fruitful feedback on my work during his visit to EPFL.

I would like to thank all the current and former members of the DATA lab at EPFL, Abdallah Elguindy, Daniel Lupei, Mohammad Dashti, Aleksandar Vitorovic, Amir Shaikhha, Yannis Klonatos, Milos Nikolic, Vojin Jovanovic, Thierry Coppey, Immanuel Trummer, Lionel Parreaux, Andrej Spielmann, and Andres Notzli, who provided me with valuable feedback and support on my research. I am also grateful to my colleagues, Matt Olma, Iraklis Psaroudakis, Danica Porobic, and Renata Borovica, for the friendly support during the stressful times. I would also like to thank Simone Muller for being the backbone of our lab.

I am deeply grateful for the support and encouragement of my friends and family. I would like to thank my best friend, Eslam Elnikety who has a big role in shaping my life. I would also like to thank the women in my life, who continuously supported me, i.e., my mom Soad, my sisters, Sara and Soha, and my partner Aida have always been there for me. I am thankful to my friends Mirena Dimova and Christina Strelchouk for supporting me and being patient during the stressful periods of my Ph.D.

This work was supported by the EPFL DATA lab, the ERC grant 279804.

Lausanne, 28 March 2017

M. E.



Abstract

With the advent of emerging technologies and the Internet of Things, the importance of on-line data analytics has become more pronounced. Businesses and companies are adopting approaches that provide responsive analytics to stay competitive in the global marketplace. Online analytics allow data analysts to promptly react to patterns or to gain preliminary insights from early results that aid in research, decision making, and effective strategy planning. The growth of data-velocity in a variety of domains including, high-frequency trading, social networks, infrastructure monitoring, and advertising require adopting online engines that can efficiently process continuous streams of data.

This thesis presents foundations, techniques, and systems' design that extend the state-of-the-art in online query processing to efficiently support relational joins with arbitrary join-predicates (beyond traditional equi-joins); and to support other data models (beyond relational) that target machine learning and graph computations. The thesis is divided into two parts:

We first present a brief overview of Squall, our open-source online query processing engine that supports SQL-like queries on top of streams. Then, we focus on extending Squall to support efficient theta-join processing. Scalable distributed join processing requires a partitioning policy that evenly distributes the processing load while minimizing the size of maintained state and duplicated messages. Efficient load-balance demands apriori-statistics which are not available in the online setting. We propose a novel operator that continuously adjusts itself to the data dynamics, through adaptive dataflow routing and state repartitioning. It is also resilient to data-skew, maintains high throughput rates, avoids blocking during state repartitioning, and behaves as a black-box dataflow operator with provable performance guarantees. Our evaluation demonstrates that the proposed operator outperforms the state-of-the-art static partitioning schemes in resource utilization, throughput, and execution time up to 7x. In the second part, we present a novel framework that supports the Incremental View Maintenance (IVM) of workloads expressed as linear algebra programs. Linear algebra represents a concrete substrate for advanced analytical tasks including, machine learning, scientific computation, and graph algorithms. Previous works on relational calculus IVM are not applicable to matrix algebra workloads. This is because a single entry change to an

Acknowledgements

input-matrix results in changes all over the intermediate views, rendering IVM useless in comparison to re-evaluation. We present Lago, a unified modular compiler framework that supports the IVM of a broad class of linear algebra programs. Lago automatically derives and optimizes incremental trigger programs of analytical computations, while freeing the user from erroneous manual derivations, low-level implementation details, and performance tuning. We present a novel technique that captures Δ changes as low-rank matrices. Low-rank matrices are representable in a compressed factored form that enables cheaper computations. Lago automatically propagates the factored representation across program statements to derive an efficient trigger program. Moreover, Lago extends its support to other domains that use different semi-ring configurations, e.g., graph applications. Our evaluation results demonstrate orders of magnitude (10x—100x) better response times in favor of derived trigger programs in comparison to simple re-evaluation.

Key words: online query engine, theta-joins, efficient joins, skew-resilience, adaptivity, matrix algebra, incremental view maintenance, rewrite systems, incremental computation, compiler optimization, graph computation



Résumé

Avec l'avènement des nouvelles technologies et l'internet des objets, l'importance des analyses de données en ligne a grandi. Les entreprises adoptent des approches qui fournissent des analyses réactives afin de rester compétitive sur le marché global. L'analyse de données en ligne permettent aux analystes de répondre promptement à des tendances ou acquérir une certaine prévision à partir de premier résultat aidant la recherche, la prise de décisions et la mise en place de stratégies efficaces. La vitesse croissante des données dans une variété de domaines tel que le trading à haute fréquence, les réseaux sociaux, le monitoring d'infrastructure et la publicité nécessite l'adoption de moteur de recherche pouvant analyser efficacement un flot de données continu.

Cette thèse présente des fondements, des techniques et des conceptions de systèmes qui repousse l'état de l'art dans le traitement des requêtes en ligne afin de soutenir efficacement des join relationnel avec des join-prédicats (au delà des traditionnels equi-joins) et soutenir d'autres modèles de données qui cible le machine learning et le calcul de graphes. Cette thèse est divisée en deux parties :

Nous présentons d'abord un court survol de Squall, notre moteur de traitement de requêtes en ligne, une source ouverte qui peut traiter les requêtes du genre SQL en plus des flux de données. Ensuite, on se concentre sur l'expansion de Squall afin de supporter efficacement le traitement theta-join. Le traitement extensible de join distribués nécessite une politique de partitionnement qui distribue de manière égale la charge de traitement tout en minimisant la taille de l'état maintenu et des messages dupliqués. Afin d'équilibrer efficacement la charge il faut des statistiques en amont qui ne sont pas disponible sur le réglage en ligne. Nous proposons un opérateur novateur pouvant continuellement s'ajuster à la dynamique des données à travers un routage adaptatif de celles-ci et re-partitionnement des états. Il est aussi résistant aux biais de données, maintient un rendement élevé et ne bloque pas lors du re-partitionnement. Il agit également comme une boîte noire des flots de données avec des garanties de performance vérifiable. Notre évaluation démontre que l'opérateur dépasse les modèles de partitionnement statique dans l'utilisation des ressources, le rendement et la vitesse d'exécution jusqu'à un facteur 7.

Dans la seconde partie, nous présentons une structure novatrice qui supporte le Incremental View Maintenance (IVM) des charges de travail exprimées sous la forme de programme en algèbre linéaire. L'algèbre linéaire représente un substrat concret pour des tâches d'analyses

Acknowledgements

avancées incluant, l'apprentissage des machines, le traitement de données scientifiques et les algorithmes de graphiques. Des travaux précédents sur le calcul relationnel IVM ne s'appliquent pas à une matrice de charge de travail en algèbre. Ceci est causé par le fait que la modification d'une seule entité dans une matrice-input cause un changement dans tous les points de vues intermédiaires, rendant l'IVM inutile comparée à la ré-évaluation. Nous présentons donc Lago, une structure modulaire unifiée pouvant supporter les IVM d'une large classe de programmes en algèbre linéaire. Lago déduit et optimise automatiquement des programmes d'analyse progressive, tout en libérant l'utilisateur d'erreur de déduction manuelle, des petits détails de mise en œuvre et le réglage de performance. Nous présentons une technique novatrice qui capture les changements Δ sous forme de matrices à rang inférieur. Les matrices à rang inférieur sont représentables dans une version compressée qui permet un traitement moins coûteux. Lago propage automatiquement les représentations factorisées dans les affirmations du programme afin d'en dériver un trigger program qui est efficace. De plus, Lago étend son domaine d'action à d'autres applications de configuration semi-ring comme par exemple les applications de graphes. Nos résultats d'évaluation montrent un temps de réponse de 10 à 100 fois plus rapide en faveur des trigger programs déduits en comparaison à la simple ré-évaluation.

Mots clefs : moteur de requête en ligne, theta-joins, joins efficace, résistance aux biais, adaptativité, algèbre matricielle, incremental view maintenance, système de réécriture, traitement progressif, optimisation de compilation, calcul de graphes.



Zusammenfassung

Mit dem Aufkommen neuer Technologien und dem Internet der Dinge ist die Bedeutung der Online-Datenanalytik ausgeprägter geworden. Geschäfte und Unternehmen nehmen Ansätze an, die responsive Analytik bieten, um auf dem globalen Markt wettbewerbsfähig zu bleiben. Online Analytik erlaubt es Datenanalytikern, zeitnah auf Muster zu reagieren oder vorläufige Erkenntnisse aus früheren Ergebnissen zu gewinnen, die bei der Forschung, Entscheidungsfindung und einer effektiven Strategieplanung helfen. Das Wachstum der Datengeschwindigkeit in einer Vielzahl von Domänen, einschließlich Hochfrequenzhandel, soziale Netzwerke, Infrastruktur Überwachung und Werbung, erfordern Online Motoren, die effizient kontinuierliche Datenströme verarbeiten können. In dieser Arbeit werden Grundlagen, Techniken und Systemdesign, die den Stand der Technik in der Online-Abfrageverarbeitung erweitern, um relationale Joins mit beliebigen Join-Prädikaten (über traditioneller Equi-Joins hinaus) effizient zu unterstützen; und um andere, nicht relationale Datenmodelle, die auf maschinelles Lernen und Graphentheorie zielen, zu unterstützen. Die Arbeit ist in zwei Teile gegliedert: Zuerst präsentieren wir einen kurzen Überblick über Squall, unseren Open-Source-Online-Abfrage-Verarbeitungs-Engine, der SQL-ähnliche Abfragen auf Datenströme unterstützt. Dann konzentrieren wir uns auf die Erweiterung von Squall, um eine effiziente Theta-Join-Verarbeitung zu unterstützen. Die skalierbare verteilte Join-Verarbeitung erfordert eine Partitionierung Richtlinie, die die Verarbeitungslast gleichmäßig verteilt, während die Größe des beibehaltenen Zustandes und der duplizierten Nachrichten minimiert wird. Effiziente Lastverteilung fordert apriori-Statistiken, die im Online Einstellung nicht verfügbar sind. Wir schlagen einen neuartigen Operator vor, der, durch adaptives Datenfluss-Routing und Zustands Repartitionierung, sich kontinuierlich an die Datendynamik anpasst,. Er ist auch stabil gegen Daten-Skew, hält hohe Durchsatzraten, vermeidet Blockierung während der Repartitionierung und verhält sich wie ein Black-Box-Datenfluss-Operator mit nachweisbaren Leistungsgarantien. Unsere Auswertung zeigt, dass der vorgeschlagene Operator den aktuellen Stand hinsichtlich der statischen Partitionierungsschemata in Ressourcennutzung, Durchsatz und Ausführungszeit bis zu 7x übertrifft. Im zweiten Teil präsentieren wir einen neuen Rahmen, der die inkrementelle Viewpflege (Incremental View Maintenance - IVM) von Arbeitsbelastung unterstützt, ausgedrückt als lineare Algebra-Programme. Lineare Algebra stellt eine konkrete Grundlage für fortgeschrittene analytische Aufgaben dar, einschließlich maschinelles Lernen, wissenschaftliche Berechnungen und Graphenalgorithmen. Bisherige Arbeiten zum Kalkül IVM sind nicht auf Matrix Algebra

Acknowledgements

Arbeitsbelastung anwendbar. Dies liegt daran, dass ein einziger Änderungseintrag in eine Eingabematrix zu Veränderungen über alle Zwischenansichten führt, was IVM im Vergleich zur Neubewertung nutzlos macht. Wir präsentieren Lago, ein einheitliches modulares Compiler-Framework, das das IVM einer breiten Klasse von linearen Algebra-Programmen unterstützt. Lago leitet und optimiert automatisch inkrementelle Trigger-Programme von analytischen Berechnungen ab und befreit dabei den Benutzer von fehlerhaften manuellen Ableitungen, Low-Level-Implementierungsdetails und Performance-Tuning. Wir stellen eine neuartige Technik vor, die Δ Veränderungen als niederrangige Matrizen erfasst. Niederrangige Matrizen sind in einer komprimierten Formfaktor darstellbar, die Berechnungen mit weniger Aufwand ermöglicht. Lago propagiert automatisch die faktorisierte Darstellung über die Programmanweisungen, um ein effizientes Auslöserprogramm abzuleiten. Außerdem erstreckt sich Lago seine Unterstützung für andere Anwendungen und Domänen unterschiedlicher Halbring Konfigurationen, z. B. Graph Anwendungen. Unsere Auswertungsergebnisse zeigen in Größenordnung (10x—100x) bessere Reaktionszeiten zugunsten abgeleiteter Triggerprogramme im Vergleich zur einfachen Neubewertung.

Schlüsselwörter: Online-Abfrage-Motor, Theta-Joins, effiziente Joins, Skew-Resilience, Adaptivität, Matrix-Algebra, inkrementelle Viewpflege, Rewrite-Systeme, inkrementelle Berechnung, Compiler-Optimierung, Graphentheorie

Contents

| | |
|---|-------------|
| Acknowledgements | i |
| Abstract (English/Français/Deutsch) | iii |
| List of figures | xiii |
| List of tables | xv |
| 1 Introduction | 1 |
| 1.1 Existing Systems and Limitations | 2 |
| 1.2 Contributions and Thesis Outline | 4 |
| 2 Background: A Bird's Eye View of Online Processing | 7 |
| 2.1 Introduction | 7 |
| 2.2 Requirements for Online Systems | 10 |
| 2.3 The Online Stream Processing Terrain | 12 |
| 2.4 Summary | 19 |
| 3 Squall: Online Query Processing | 21 |
| 3.1 Data Model | 21 |
| 3.2 The Squall Framework | 22 |
| 3.2.1 Interface | 23 |
| 3.2.2 Query plans | 24 |
| 3.2.3 Operators | 25 |
| 3.2.4 Query optimizer | 26 |
| 3.2.5 Underlying Processing Platform | 26 |
| 3.3 Summary | 27 |
| 4 Online Theta Joins | 29 |
| 4.1 Challenges and Contributions | 30 |
| 4.2 Background & Preliminaries | 31 |
| 4.2.1 Join Partitioning Scheme | 31 |
| | ix |

Contents

| | | |
|----------|---|------------|
| 4.2.2 | Operator Structure | 32 |
| 4.2.3 | Input-Load Factor (ILF) | 33 |
| 4.2.4 | Grid-Layout Partitioning Scheme | 34 |
| 4.3 | Related Work | 35 |
| 4.4 | Intra-Operator Adaptivity | 37 |
| 4.4.1 | Monitoring Statistics | 38 |
| 4.4.2 | Analysis and Planning | 38 |
| 4.4.3 | Actuation | 49 |
| 4.4.4 | Equi-Joins Specialization | 54 |
| 4.5 | Evaluation | 56 |
| 4.5.1 | Skew Resilience | 58 |
| 4.5.2 | Performance Evaluation | 58 |
| 4.5.3 | Scalability Results | 62 |
| 4.5.4 | Data Dynamics | 63 |
| 4.5.5 | Summary | 64 |
| 5 | Lago: Online Advanced Analytics | 65 |
| 5.1 | Challenges and Contributions | 66 |
| 5.2 | Incremental Computation Δ | 67 |
| 5.2.1 | The Delta Δ Representation | 69 |
| 5.3 | The LAGO Framework | 71 |
| 5.3.1 | Architecture | 72 |
| 5.3.2 | Lago DSL | 73 |
| 5.3.3 | Transformation Rules | 77 |
| 5.3.4 | Meta-Information | 81 |
| 5.3.5 | Wiring it all together | 84 |
| 5.4 | Other Use cases | 86 |
| 5.4.1 | Incremental Linear Regression | 86 |
| 5.4.2 | Incremental Matrix Powers | 87 |
| 5.5 | Related Work | 93 |
| 5.6 | Evaluation | 95 |
| 5.6.1 | Incremental Linear Regression | 96 |
| 5.6.2 | Graph Analytics | 99 |
| 5.6.3 | Scalability Evaluation | 103 |
| 6 | Conclusion | 105 |
| A | Appendix | 107 |
| A.1 | Analysis under Window Semantics | 107 |
| A.2 | LAGO Rules | 110 |

Contents

| | |
|-------------------------|------------|
| Bibliography | 125 |
| Curriculum Vitae | 127 |

List of Figures

| | | |
|------|--|----|
| 3.1 | The Squall query processing engine. An example query plan is first translated to a logical plan, then to a physical plan and finally to a storm topology. | 22 |
| 4.1 | Examples of the join-matrix \mathcal{M} for various (monotonic) joins $R \bowtie_{\theta} S$ between two streams (relations) R and S | 31 |
| 4.2 | (a) $R \bowtie_{\theta} S$ join-matrix example, grey cells satisfy the $\theta \neq$ predicate. (b) a (2,4)-mapping scheme using $J = 8$ machines. (c) the theta-join operator structure. 32 | |
| 4.3 | (a) join-matrix with dimensions 1GB and 64GB (b) a (8,8)-mapping scheme assigns an ILF of $(8\frac{1}{8})$ GB (c) a (1,64)-mapping scheme assigns an ILF of 2GB. . . | 33 |
| 4.4 | Migration from a (8,2)- to a (4,4)-mapping. | 42 |
| 4.5 | (a) decomposing $J = 20$ machines into independent groups of 16 and 4 machines. 43 | |
| 4.6 | Theta-joins elasticity. | 47 |
| 4.7 | Execution time performance results part I. | 59 |
| 4.8 | Execution time performance results part II. | 59 |
| 4.9 | Operator metrics performance results part I. | 60 |
| 4.10 | Operator metrics performance results part II. | 60 |
| 4.11 | Scalability performance results. | 62 |
| 4.12 | Performance results under fluctuations. | 63 |
| 5.1 | Deriving the trigger program for the matrix powers program A^8 | 68 |
| 5.2 | Propagation of data-changes in matrix programs. | 69 |
| 5.3 | The architecture of the Lago framework. | 71 |
| 5.4 | The core Lago DSL divided into two main classes, i.e., matrix and scalar operations. 73 | |
| 5.5 | Syntactic sugar: Examples of additional operations defined using compositions of the Lago DSL. | 75 |
| 5.6 | Matrix addition and multiplication in the core Lago DSL generalized for semirings 76 | |
| 5.7 | Program \mathcal{P} represents all-pairs Graph Reachability or Shortest Path after k-hops depending on the semiring configuration. | 76 |

List of Figures

| | | |
|------|---|-----|
| 5.8 | Delta Δ derivation rules for the core language constructs. The <code>iterate</code> construct is first unfolded using the simplification rules in the appendix before applying Δ rules on it. Moreover, the Δ rule for matrix inverse enables the cheaper Woodbury formula as explained in the subsequent examples in section 5.4.1. | 77 |
| 5.9 | Inferring dimensions and cost of matrices. | 81 |
| 5.10 | An example for bottom-up propagation of meta-information. | 83 |
| 5.11 | Lago IVM phases. | 84 |
| 5.12 | Walking through an example undergoing the IVM phases. | 85 |
| 5.13 | Step-by-step Δ derivation of the Ordinary Least Squares program till the factorization phase. | 89 |
| 5.14 | Step-by-step Δ derivation of Matrix powers till the factorization phase. | 92 |
| 5.15 | Performance evaluation of Incremental Linear Regression. | 96 |
| 5.16 | Performance Evaluation of Meta-Information specialization opportunities. . . | 101 |
| 5.17 | Performance Evaluation of Incremental Graph programs using <code>Symbit</code> for Reachability and <code>Dense</code> for Shortest Distance. | 102 |
| 5.18 | Scalability and additional storage results. | 103 |
| A.1 | Simplification rules _____ | 110 |
| A.2 | Equivalence rules _____ | 111 |
| A.3 | Inferring symmetry of matrices. _____ | 111 |
| A.4 | Inferring Sparse Structures of matrices. _____ | 111 |
| A.5 | Inferring Ranks of matrices. _____ | 111 |
| A.6 | Inferring structure of Triangular matrices (\mathcal{U} : Upper triangle, \mathcal{L} : Lower triangle) . | 111 |
| A.7 | Inferring layout of bit vector matrices (\mathcal{R} : Row layout, \mathcal{C} : Column layout) ____ . | 111 |

List of Tables

| | | |
|-----|---|-----|
| 4.1 | Queries used for Theta-join evaluation. | 56 |
| 4.2 | Query performance under skew. | 58 |
| 5.1 | Equivalent operations in Matlab, R, and Lago. | 74 |
| 5.2 | Report on compilation metrics. | 97 |
| 5.3 | The average Octave and Spark view refresh times in seconds for INCR of P^{16} and a batch of 1,000 updates. The row update frequency is drawn from a Zipf distribution. | 104 |

1 Introduction

This thesis presents foundations, techniques, and system designs that extend the state-of-the-art in online query processing to efficiently support relational joins with arbitrary join-predicates (beyond traditional equi-joins); and to support other data models that target machine learning, scientific, and graph computations.

With the advent of emerging technologies and data acquisition tools, the importance of data analytics has become more pronounced. More than ever, businesses, companies, and institutions are incorporating data analysis tools into their workflows. Such tools provide useful information, patterns, and insights that aid in research, decision making, and effective strategy planning.

We currently live in the data deluge era where data grows consistently. In general, data continuously expands in three main dimensions, namely volume, variety, and velocity [114]. For instance, Facebook maintains a 300PB data warehouse with a daily data-growth rate of 600TB [172]. The Large Hadron Collider (LHC) generates 30PB of raw data every year, which is used for scientific simulations [2]. Therefore, modern applications not only have to keep up with the expansion pace of data diversity and size, but they also have to be agile enough to cope with the rapid change in evolving datasets. The immense growth of data velocity in massive data domains including high-frequency trading, social networks, online gaming, infrastructure monitoring, recommendation systems, and advertisement require adopting online engines that can efficiently process continuous streams of real-time data (along with traditional batch processing systems for historical data). Businesses must embrace approaches that provide responsive analytics to stay competitive in the global marketplace. Online and responsive analytics allow data analysts and statisticians to promptly react to patterns or to gain preliminary insights from approximate or incomplete results at very early stages of the computation. The rise of various systems and solutions that process real-time data streams reflects the current trends and interests in streaming data analytics. This trend has fostered a

number of open-source and commercial frameworks for online stream processing. However, these systems lack the support for a wide class of online analytics that enable advanced analyses including general-join processing and domain-specific operations.

1.1 Existing Systems and Limitations

Michael Stonebraker *et al.* [164] present eight requirements for achieving efficient online stream processing (Chapter 2). We focus here on the main requirements for modern online analytics while discussing the limitations of existing systems. We first give an example to demonstrate these requirements.

Example. Due to the increasing ubiquity of smart devices that contain GPS functionality, many geosocial networks have emerged. Geosocial networks, such as FourSquare ⁱ, Facebook Places ⁱⁱ, Google Latitude, and Waze ⁱⁱⁱ, have millions of active users. These companies ingest large volumes of high-speed data including sensor data, geolocation information, and crowdsourced user reports, e.g., accidents, traffic jams, and nearby police units. Running online analytics on the ingested data helps companies provide real-time services such as tracking friends [53, 69], identifying mobility patterns, avoiding traffic congestions, recommending routes and deployment of police units, and improving road safety [75]. To provide these services on geospatial datastreams, an efficient online analytics engine should satisfy a set of requirements: a) ensure low-latency and responsive processing, b) scale out across commodity-hardware machines to distribute the large volumes of ingested data, and c) support expressible analytics on diverse data models. For example, merging different streams of sensory-data using complex join conditions, e.g., spatial or similarity joins; building progressive machine learning models to predict traffic congestion and recommend routes; and maintaining dynamic social graphs to study network structure.

Support for low latency responsive analytics. One of the main requirements for online analytics to provide fast responses to queries as more data arrives. Many current systems for data analytics have been developed to tackle data analysis at a large scale, including relational database management systems [122], multidimensional OLAP engines [44], MapReduce [56] and Spark [188] for general-purpose processing and other specialized solutions for graph computations [78, 105, 38, 173], array processing [113, 163, 119, 123, 80], data mining and machine learning [131, 78]. These systems are geared towards offline batch processing, i.e., they compute high throughput analytics on fixed static datasets. One way to support dynamically evolving datasets is to re-evaluate computations every time a batch of input tuples arrive. However, recomputing offline data analytics from scratch on every data change of small or

ⁱwww.foursquare.com

ⁱⁱwww.facebook.com/places/

ⁱⁱⁱwww.waze.com

moderate size is highly inefficient.

Online stream processing engines such as Borealis [7] and Stream [21] are designed for low-latency query processing on data streams. They provide declarative languages that support SQL-like queries (e.g., selection, projection, joins, etc) on fixed windows of data streams. These systems fulfil the low latency requirement. However, their utility is limited to relational processing and they do not support data parallelism, which results in limited scalability and the inability to handle long-lived data (large states), i.e., small-window semantics.

Support for stateful and scalable analytics. Another important requirement for online analytics is the support for large scale stateful processing [39]. Modern online applications typically experience high input rates of streaming data. It is necessary to provide platforms and systems that can scale out to accommodate these requirements. Recently, several distributed stream processing engines have emerged to handle the increasing volumes of data streams including, Storm [127], Heron [110], Spark Streaming [190, 189], and Flink [20]. Such systems leverage data partitioning and parallelism to distribute work among a cluster of machines.

These frameworks have several limitations. First, most of these systems provide frameworks for general online data processing. This puts the burden of building efficient query plans or engines on the developer. Second, some of them provide an interface that supports simple relational query processing based on stateless operators, e.g., projections, selections, and joins with static databases^{iv}. On the other hand, the more interesting and advanced operations are stateful, e.g., streaming joins [39, 21]. However, scaling-out stateful operations is more challenging as it entails careful partitioning of state and maintaining an even load distribution. Moreover, the online setting necessitates delicate and adaptive partitioning mechanisms to ensure load balancing at all times. In Chapter 4, we present a provably efficient and adaptive online operator for stateful theta-join processing.

Support for queryable and expressive analytics on top of streams. In the quest for better insights, modern applications demand for advanced analytics like machine learning, scientific computing, and graph processing. Data processing systems that support domain-specific operations can greatly empower users to perform complex data analysis. Current online systems are suitable for relational and descriptive analytics which mainly evaluates simple equi-joins and aggregations on the ingested data. They do not provide the infrastructure to support online analytics of other data models beyond relational. In Chapter 5, we present a framework that supports incremental evaluation of matrix programs that can capture machine learning and graph computations.

^{iv}Flink is an exception as it supports stateful window semantics

1.2 Contributions and Thesis Outline

Chapter 2 gives an overview of the online stream processing landscape, including the various classes of online processing and existing systems. After that, we present foundations, techniques, and system designs that support online incremental processing of advanced analytics within the context of two different systems:

Squall: Chapter 3 gives a brief summary of our open source online query processing engine Squall that supports SQL-like computations on top of streams. Then, Chapter 4 describes how to extend online relational analytics (Squall) with support for efficient and general join-processing. In particular, scalable join processing in a distributed environment requires a partitioning policy that evenly distributes the processing load while minimizing the size of state maintained and number of messages communicated. In an online or streaming environment in which no statistics about the workload are known, we show how traditional static partitioning approaches perform poorly. We present a novel parallel online dataflow join operator that supports arbitrary join predicates. The proposed operator continuously adjusts itself to the data dynamics through adaptive dataflow routing and state repartitioning. The operator is resilient to data skew, maintains high throughput rates, avoids blocking behavior during state repartitioning, takes an eventual consistency approach for maintaining its local state, and behaves strongly consistently as a black-box dataflow operator. We prove that the operator ensures a constant competitive ratio of 1.25 in data distribution optimality and that the cost of processing an input tuple is amortized constant, taking into account adaptivity costs. Our evaluation demonstrates that our operator outperforms the state-of-the-art static partitioning schemes up to 7x in resource utilization, throughput, and execution time.

Lago: The second part of the thesis targets efficient online evaluation of matrix algebra programs. Statistical models, machine learning applications, and graph algorithms are usually expressed as linear algebra programs which is beyond the relational data model. There exists many systems and frameworks [131, 78, 113, 123, 80, 119, 149, 160, 192] that optimize such programs under large volumes of offline data. Under the online setting, the re-evaluation of the analytic programs on each matrix change is prohibitively expensive. We present Lago, a unified modular compiler framework that supports the IVM of a broad class of linear algebra programs. Lago automatically derives and optimizes incremental trigger programs of analytical computations, while freeing the user from erroneous manual derivations, low-level implementation details, and performance tuning. We present a novel technique that captures Δ changes as low-rank matrices (Section 5.2.1). Low-rank matrices are representable in a compressed factored form that enables converting programs that utilize expensive $\mathcal{O}(n^3)$ matrix operations, e.g., matrix-matrix multiplication and matrix-inverse, to trigger programs that evaluate delta expressions with asymptotically cheaper $\mathcal{O}(n^2)$ matrix operations, e.g., matrix-vector multiplication. Lago utilizes the low-rank property and automatically propagates it

across program statements to derive an efficient trigger program. Moreover, Lago extends its support to other applications and domains of different semi-ring configurations, e.g., graph applications. Our evaluation results demonstrate orders of magnitude (10x—100x) better response times in favor of derived trigger programs in comparison to simple re-evaluation.

This work includes material from several publications in which the author of this thesis is the lead author or a co-author.

- Chapter 4 presents material where the author led the research, design and implementation of the system and strategies. The author also participated in the core design and implementation of Squall as presented in Chapter 3.
 - *Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, Christoph Koch.*
Scalable and Adaptive Online Joins.
VLDB 2014.
 - *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.
- The author also contributed in introducing the concept of low-rank delta matrices and representing them in a compressed factored-form for incremental computation in Chapter 5 (Section 5.2). These contributions have been published in SIGMOD 2014. Moreover, the author led all the subsequent research and evaluation in Chapter 5 which presents Lago, a compiler framework for deriving and optimizing IVM trigger programs for a class of matrix algebra programs. Stefan Mihaila and Daniel Espino contributed to the implementation of the framework as part of their master thesis research. This work is under submission towards SIGMOD 2018.
 - *Mohammed Elseidy, Amir Shaikhha, Daniel Espino, Stefan Mihaila, Christoph Koch.*
Towards Incremental Computation of Advanced Analytics.
SIGMOD 2018, under submission.
 - *Milos Nikolic, Mohammed Elseidy, Christoph Koch.*
LINVIEW: Incremental View Maintenance for Complex Analytical Queries.
SIGMOD 2014.

2 Background: A Bird's Eye View of Online Processing

This chapter serves as a brief overview of online stream processing. First, we discuss the importance of online analytics while presenting several real-world use cases and applications. Then, we present the main characteristics and requirements of efficient online systems, as presented by Michael Stonebraker *et al.* [164], to meet the demands of different stream processing applications. Finally, we give a brief survey of the online stream processing terrain where we discuss three main categories of emerging platforms: query engines, streaming algorithms, and general-purpose online systems.

2.1 Introduction

Over the last decade, Big Data has become the principal term that describes the current era of information systems. Scientists, businesses, markets, and institutions process large amounts of data to acquire insights and knowledge that help them in research and decision making. However, processing Big Data efficiently has many challenges. At first, the biggest challenge was scaling analytics to immense *volumes* of data. A large body of research has been developed over the years to tackle this challenge resulting in frameworks, algorithms, and commercial and open-source systems that can efficiently scale to massive volumes. However, recently, the requirements of Big Data have evolved to include additional challenges, in particular, the *variety* of ingested data types and models and the *velocity* of data changes. This is referred to as the three V's of the Big Data challenge [114], and are defined more precisely as:

1. Volume: the ability to analyze and process massive amounts of data, i.e., terabytes and petabytes.
2. Variety: the ability to clean and incorporate data from different sources and formats.
3. Velocity: the ability to cope with high-speed ingested data.

This thesis focuses on the challenges related to large volumes and high-velocity data. It is difficult for current traditional data processing systems to cope with the fast-paced growth of large-data domains including social networks, high-frequency trading, online gaming, and online advertisement [32, 157]. There is an ever-increasing demand for efficient analytics that process large volumes of data in an online fashion. Businesses are currently shifting towards real-time data-based products that allow continuous computations, interactivity, and instant decision-making. The following examples demonstrate several applications that require continuous online processing:

- **Smart Cities.** Smart cities [93] is an urban development vision that integrates communication technology and Internet of Things solutions in a secure fashion to manage a city's assets and to incubate a human adaptive environment. Real-time data from different sources is analyzed for city planning and human mobility [90]. For instance, the data gathered is used to help governments in the dynamic decision making process [153] such as optimizing public transport and allowing people avoid traffic congestion across different routes within a city. The urban data is also used for weather and air content monitoring.
- **Business Intelligence.** In the business sector, online analytics is widely used for inventory management, understanding customer behavior to improve the customer online experience, and evaluating sales performance in real-time to achieve sales quotas through instant incentives such as discounts, bundles, free shipping, and easy payment terms. For instance, large businesses and retailers offer specialized recommendations and promotion programs to reach potential customers. They search for patterns in customers and sales data to find suitable suggestions and proposals during an active customer session. This requires maintaining information about customers' profiles including shopping history, location information, and interests. Amazon offers product recommendations according to the session information, including the recently explored products [1]. At Twitter, many of the recommendations are based on recent tweets [110]. Furthermore, many start-ups, such as QuantCast and RocketFuel, base their businesses around online advertising.
- **The Medical Sector.** Hospitals use distributed stream processing for health monitoring objectives. For instance, they monitor patients' health through real-time streams of measurement data generated from different medical instruments and sensors [156]. This helps medical personnels in diagnosis, exploring correlations in patient diseases, and instantly reacting to proactive medical alerts.
- **Online Anomaly and Fraud Detection.** It is crucial to continuously detect fraudulent activities with credit card transactions to prevent damage and abuse. For example, online marketplace providers such as eBay and BetFair run sophisticated fraud detection algorithms on real-time trading activity. The banking sector too monitors and processes

multiple transaction streams every day to detect suspicious activities and to prevent credit card fraud [99, 95].

- **Stock Market and Algorithmic Trading.** In exchange markets, fast and responsive actions are essential for achieving profits. For instance, matching ask/bid transactions in order books requires fast and continuous processing. Moreover, acting fast during arbitrage opportunities can result in high profit gains. Arbitrage opportunities appear when a commodity is sold on one market exchange at a specific price and bought on another one at a lower price. Therefore, it is crucial to act fast given such opportunities. Not only does this require low-latency processing, but it also requires advanced analytic processing. For instance, trading systems can analyze additional data from external data providers, e.g., social networks, to improve trading strategies. For example, a positive sentiment around a particular stock (using sentiment analysis), can trigger a bullish stance towards the stock price. Combining trading strategies with social media data typically involves fast and advanced query processing.
- **Real-Time Monitoring.** Real-time surveillance and monitoring require low latency processing to take fast action in critical situations. For instance, interconnected infrastructures, such as utility grids, computer networks, and manufacturing facilities maintain and monitor their performance, availability, and capacity [108]. In other domains such as the Internet of Things (IoT) and sensor networks, data is continuously ingested and analyzed in real-time to enable interactivity and instant response to urgent situations.
- **Online Gaming.** In the gaming industry, stream processing is used to enhance the gaming experience of players. For instance, Supercell [19], a gaming company that provides online games for portable devices such as Clash of Clans and Boom Beach, uses Amazon Kinesis [18] to process data streams generated from various devices. Amazon Kinesis enables Supercell to support real-time analysis of games, improve interactive player experience, and run personalized business analytics [19].

There are endless opportunities to utilize real-time streaming data. Different applications including, web pages personalization, weather forecasting, pay-as-you-drive insurance models, recommender systems, and energy trading services are emerging domains that are beginning to shift their business models to benefit from real-time analytics. With the ubiquity of the Internet of Things, distributed real-time stream processing will soon be the de-facto standard for analytics.

2.2 Requirements for Online Systems

Streaming applications have various requirements not supported by traditional batch processing engines. In [164], Michael Stonebraker *et al.* present a set of general requirements for data stream processing engines that have become accepted for real-time streaming applications. Some of these requirements inherently conflict with each other as different tradeoffs that depend on the application semantics. We present them next:

Keep the Data Moving. Online systems, as opposed to offline engines, need to maintain *low-latency* in processing incoming data. The goal is to avoid costly operations on the critical path of online execution. To achieve that, data needs to be processed on-the-fly. In particular, every new tuple contributes a small change to the corresponding internal state and the final result. Computations must be incremental and avoid costly re-evaluation per change. State management should not cause, to the degree possible, costly storage overheads, such as writing transactional commits and logs to disk, that have a detrimental effect on processing performance.

Process and Respond Instantaneously. Sustaining *high-throughput* is a critical requirement for online systems. In particular, they should be able to ingest large volumes of data while maintaining *low-latency* response times. Stream processing engines should be equipped with highly optimized execution engines that deliver real-time responses for high-volume applications. The various components of the engine need to be designed to achieve a balance between high-throughput and low-latency. For instance, batching data tuples results in better throughputs at the cost of increased response times. Therefore, it is important to find the right batch size that meets the requirements of the online application.

Partition and Scale Applications Automatically. Online engines should be able to scale-out to accommodate the large volumes of input data and to maintain high-throughput rates. Modern applications typically require ingesting high-velocity data streams, on-the-fly intensive computations, and maintaining large state sizes [39, 21]. A single-machine configuration is not suitable for these requirements. On the other hand, distributed computation has become increasingly important given the favourable price-performance characteristics of low-cost commodity clusters. Under this setting, streaming applications can be split over multiple machines to distribute load and computation. Moreover, the processing engine could take advantage of modern multi-processors and multicore architectures to avoid blocking for external events and thereby enabling low-latency. A critical challenge in parallel and distributed platforms is to achieve load balance which ensures equal partitioning of workload and state across the available resources, thereby increasing efficiency and utilization. Therefore, efficient online processing requires automatic, transparent, and agile load balancing to efficiently respond to changes in runtime, e.g., input-data rates and data statistics. These changes might have a skewed distribution and can degrade performance severely.

- **Skew Resilience.** In statistics and probability theory, skew is the measure of asymmetry in the probability distribution of a random variable around its mean. Skewed distributions exist abundantly in the real world, ranging from natural, e.g., biological or physical systems [138, 88, 137, 121] to artificial, i.e., man-made, phenomena [194, 148, 138]. In practice, many applications [185, 184] require analyzing data that is naturally characterized by a skewed distribution. Parallel and distributed systems for data processing are highly vulnerable to data skew [111, 112]. Data skew has a direct impact on distributed query processing performance as it results in imbalanced load and overloaded nodes which limit performance and restrict scalability [185, 177]. Skew vulnerability is more pronounced in the online setting for several reasons: First, streams are liable to continuous fluctuations in input-value distribution, as can be seen in the case of concept drifts [77]. Second, the sequential access semantics of streaming data makes load balancing vulnerable to input arrival order. This vulnerability, referred to as *temporal skew*, potentially degenerates parallel computation to serial execution. A skewed load distribution has severe consequences on online performance. Overloaded nodes represent computational or I/O bottlenecks that harm latency by orders of magnitude [25]. A single overloaded machine has a ripple effect over the entire distributed plan. In particular, it can congest the network queues of online processing pipelines crippling the entire plan. In practice, a single limping node impacts the entire plan's resource utilization and overall performance [65, 64]. In this thesis, we present a data-flow join operator (chapter 4) that is content-insensitive and resilient to data skew preventing any bottlenecks during online processing.

Support for Expressive Declarative Languages. Online processing systems use continuous and long-running queries to analyze dynamic datasets. Two desirable properties of such query languages are: a) *expressiveness* where users can easily run various queries that capture complex conditions in streaming data, and b) *declarativity* where users can specify queries using high-level domain-specific languages rather than a low-level imperative programming model. Traditional query processors provide relational operators such as selections, projections, equi-joins, and SQL-like query syntax for expressing grouped aggregations over stream windows. In the quest for deeper insights, modern applications increasingly demand more powerful analytics. Online data processing engines that support both complex relational and domain-specific operations can greatly enable analysts to perform advanced analysis. This thesis presents techniques, designs, and algorithms that enable online query engines to efficiently support both relational joins with general join-predicates and incremental evaluation of matrix algebra programs. Matrices can model various domains including machine learning, graph processing, and scientific computing.

Handle Stream Imperfections. In a conventional database, data is always available before running any queries. However, in the online setting, there are no guarantees about data arrival order. An online system should include built-in mechanisms that provide resilience

against stream “imperfections” such as making provision for handling delayed, missing, or out-of-order data.

Integrate Stored and Streaming Data. Many streaming applications require comparing “present” with “past” states. It is desirable for an online engine to have the capability of combining streams with static offline data and provide careful management of stored state. An application example for this use case is fraud detection or other data mining applications that try to identify unusual activity. This can be realized by summarizing usual historical activity patterns as a “signature”, and then comparing with present activity in real-time.

Guarantee Data Safety and Availability. Distributed systems are vulnerable to failure [70] including hardware-related faults such as hard disk failures, I/O device failures, software bugs and errors, driver failures, physical damage, etc. An efficient engine should be resilient to such failures, available at all times, and capable of failure recovery while maintaining the integrity of data and state.

Generate Predictable Outcomes. Streaming engines should be able to process real-time data in a predictable manner with deterministic guarantees about the output. Moreover, the ability to generate deterministic results is an important requirement for fault tolerance and recovery. This requirement stems out from the fact that reprocessing the same input stream should yield the same predictable outcome regardless of when it is executed.

2.3 The Online Stream Processing Terrain

This section provides a brief overview of the online processing landscape where we discuss classes of online computation and existing systems for online processing. For a detailed discussion about the different online systems, we refer the interested reader to Liu’s comprehensive survey [118].

MapReduce Systems

The MapReduce framework provides an efficient distributed computational model for large volumes of static data. The framework is designed to support high-throughput batch processing. However, MapReduce batch processing systems [56, 5, 100] are not amenable to online and low-latency processing because the framework is built on top of blocking components. In particular, a MapReduce job does not produce any output results before all the input data has been processed. A job consists of map stage followed by a reduce stage. The reduce stage only begins after all the mappers finish processing their input data. If the computation consists of multiple dependent MapReduce jobs, subsequent jobs do not begin until the previous is done [72]. This framework is not designed for low-latency and responsive computations.

Combination of Offline and Online Systems

The Lambda Architecture proposed by Nathan Marz [128] defines a framework that runs applications on top of a fault-tolerant batch processing engine simultaneously with a low-latency online processing engine. The architecture consists of three layers. The batch layer computes views on the statically ingested data and repeats the computations periodically. By the time the output results are generated, they would be outdated, as new data has arrived in the meantime. A parallel high-speed processing layer closes this gap by simultaneously processing the new data with weaker guarantees. Note that, once the results from the batch processing layer are produced, they overwrite the corresponding preprocessed results from the speed layer. A serving layer is responsible for answering queries by merging precomputed results from both the batch and speed layers to produce an appropriate final result.

Twitter's Summingbird [34] adopts the Lambda architecture and offers a high level declarative language interface for both offline and online processing. Applications written in Summingbird can generate MapReduce jobs using Scalding [4] for offline processing. They can also generate online Storm [127] topologies for the same application. Summingbird also allows running the same application in both backends simultaneously, known as the hybrid mode. Similarly, Google offers the DataFlow [15] framework, which supports the lambda architecture. In particular, it supports both the FlumeJava and the MapReduce frameworks for offline processing and the MillWheel [14] framework for online processing.

Mini-Batch Systems

Previous works propose alternative approaches to enable online processing by modifying the Hadoopⁱ framework by eliminating its blocking behavior. For instance, the Hadoop Online Prototype (HOP) [52] and Scalla [116, 115] are systems that adopt this approach. HOP allows pipelining the intermediate data between Map and Reduce stages. It also supports pipelining data between consecutive MapReduce jobs. HOP pipelines the map output in small batches to the reducers while performing multi-pass sort-merge during reduce.

Later on, Boduo Li *et al.* [116] showed that HOP is not suitable for high-performance online processing. The reason is the inefficiency of sort-merge that imposes long stalling blocking costs and impedes incremental online processing. Sort-merge is a fundamental operation within the Hadoop framework and is widely used in partitioning and parallel processing. On the other hand, the authors propose Scalla [116, 115], a system that uses hashing to facilitate fast in-memory processing. Scalla introduces better performance in the case of memory overflows by carefully partitioning tuples among memory and disk. Both HOP [52] and Scalla [116, 115] leverage general purpose mini-batch MapReduce processing.

ⁱAn open source implementation of the MapReduce framework

There are other systems that attempt to support online analytics for batch processing engines. Spark [188] is an in-memory MapReduce system where computations are expressed as transformations on resilient distributed datasets (RDDs). An RDD is a distributed in-memory datastructure that ensures fault tolerance in the case of machine failures. Spark Streaming [190, 189] extends Spark to support online processing by introducing discretized streams, i.e., a stream of RDDs. Each RDD is amenable to transformations and processing. Spark Stream discretizes the input data into small batches of RDDs to simulate a data stream.

All the previous systems [52, 116, 115, 190, 189] alter batch processing frameworks to allow mini-batching or micro-batching. Mini-batch systems achieve better latencies than batch systems however, they still suffer from high synchronization overheads. This is because the system needs to synchronize after each processed batch and the fresh tuples are buffered until the current batch is processed. This is equivalent to a coarse-grained lock-step which requires synchronization and increased latencies. Thus, the slowest machine (staggler) limits the entire dataflow execution. A single machine can experience performance degradations for various reasons, including data skew or unexpected reasons such as small glitches in the network or limping hardware.

Online Stream Processing

The field of online stream processing can be divided into three main categories [32]: a) the online algorithmic research: sampling, synopsis, and sketch based algorithms for approximate probabilistic processing usually in a single-pass; b) the query-based online systems that have emerged from database research which this thesis contributes to; c) finally, general purpose streaming platforms for implementing and executing custom streaming applications and which this thesis builds its systems on. These different areas naturally intersect and benefit from each other.

1. Online Algorithmic Research. Online algorithmic research studies the different algorithmic aspects of computing approximate results on unbounded data streams. This is different from online aggregation [92] that operates on static databases. Many of the algorithmic problems require approximate estimates given very limited resources, i.e., processing power, storage, and main memory. That line of work presents fundamental algorithms for many problems ranging from counting [54, 23, 96] to maintaining approximate statistics, sketches, quantiles and synopsis over streams [86, 125]. Moreover, previous works present online learning methods that are capable of incrementally training models for prediction [29, 17, 67], e.g., naïve bayes predictors and Hoefding tree classifiers, or clustering tasks [10, 12, 27, 43, 48], e.g., k-means clustering. This class of computations are specialized to specific problems and algorithms and is orthogonal to our work in this thesis which supports different data models for online computation.

2. Query-based Systems. These are systems that expose a high-level SQL-like query language for online analytics on top of streaming data. There are three main classes of online query processing, in particular: a) window-stream processing, b) incremental view maintenance, c) and online aggregation. In the next chapter, we present a brief overview of Squall, an online query processing framework that supports these classes of computation. For a detailed discussion about integrating these classes of computation into the Squall framework, the interested reader can consult [175].

- **Window-stream processing.** This class of computations focuses on processing large unbounded streams of data using bounded memory resources. Window semantics [103] refers to evaluating queries on a window of recently-arrived tuples which can be either time-based or tuple-based. A window can be *sliding* where a portion of the window expires at a given time or *tumbling* where all the tuples in the window expire at the same time. There are other non-exact approaches that handle large spikes in the input data-rates by discarding tuples, i.e., load shedding. Early query engines such as TelegraphCQ [42], NiagaraCQ [46], Aurora [8, 49], Borealis [7] and STREAM [21] are designed for low-latency query processing on data streams. They provide declarative languages that support SQL-like relational operations (e.g., selection, projection, joins, etc) on fixed windows of data streams. However, they have limited scalability and are incapable of handling long-lived data, i.e. large state [39, 21].

STREAM [21] was developed at Stanford university. It was initially built to target stream environments with fluctuating load characteristics. Therefore, it was designed to adaptively work under severe resource constraints during runtime. STREAM provides a declarative language called CQL to define continuous queries on top of streams. The STREAM runtime engine provides a set of performance optimizations including synopsis sharing that materializes nearly identical synopses and sketches; global operator scheduling that reduces memory utilization in the case of bursty input streams; monitoring and adaptive query processing that collects runtime statistics and uses it to re-optimize the query plan. The early STREAM prototype did not include support for distributed processing and fault tolerance.

TelegraphCQ [42] is one of the earliest stream processing systems and has been developed at the university of Berkeley. Its design and implementation are built upon PostgreSQL [147] and Telegraph [3], an early engine for adaptive dataflow processing. TelegraphCQ was designed to support continuous queries on both relational tables and streams. It provides a declarative language called StreaQuel that supports SQL-like operations with window semantics. StreaQuel query plans are executed on Telegraph's distributed runtime environment which is equipped with adaptive logic that is used to efficiently route data across operators and distributed runtime nodes.

NiagaraCQ [46] is an online system that targets supporting continuous query processing over multiple XML files. It is the streaming processing sub-system of the Niagara project

which focuses on querying the internet. NiagaraCQ addresses scalability by taking advantage of the fact many web based queries share similar structures. It scales in the number of queries by proposing techniques for grouping continuous queries for efficient evaluation. Grouping similar structures can save on the computation cost, memory cost, and I/O cost. It also proposes providing partial results to long-running queries, where it is acceptable to provide an answer over some portion of the input data.

Aurora [8, 49] is an early stream processing engine developed by Brown university and MIT. Aurora was designed as a centralized stream processing engine for the single-machine setting and thus it does not provide advanced features such as scalability, fault tolerance, and reliability. It can run stream queries on top of unbounded streams using the Stream Query Algebra call SQuAl. The algebra supports both window and historical semantics. Moreover, Aurora also supports load shedding features which drop random tuples during overload and contention periods.

Later on, Borealis [7] was introduced as a the successor of the Aurora system. Borealis combines both, the core stream processing model of Aurora and the distributed functionality of Medusa [152]. On top of the Aurora functionality, Borealis provides several key enhancements: a) Distributed computation via inter-operator parallelism, b) a mechanism of fault tolerance to provide reliability and availability in case of system failures, c) a revision processing mechanism that handles stream and tuple imperfections, and d) a dynamic query modification mechanism that permits modifying queries during runtime.

- **Incremental View Maintenance.** IVM [30, 107, 87] stores query (intermediate) results as materialized views which are continuously updated as tuples are fed in. The goal of classical IVM is to avoid full query re-evaluation after every update. IVM relies on reusing precomputed results (views) from before. It avoids re-evaluation by only computing the delta expressions, after which the corresponding views are updated appropriately. Most notably, DBToaster [107] achieves orders of magnitude better performance on SQL queries in comparison to traditional re-evaluation through recursive IVM. In this thesis, we extend the horizon of IVM, in particular, by supporting the Incremental View Maintenance of matrix-algebra programs that can model a class of machine learning and graph computations.
- **Online Aggregation.** OA [89, 92, 102, 142] presents approaches that compute approximate aggregates of query results long before the final result is computed. OA uses statistical estimation theory tools to provide approximate results defined within confidence error bounds. It operates on static databases where data is known *ahead of time* and as more data is processed, the approximate estimate gets closer to the final result. Previous work presents novel aggregation estimation and sampling techniques from base relations (or intermediate results) to produce approximate aggregates that converge relatively quickly with tighter error bounds. This thesis is orthogonal to this line of work as we target exact

computation rather than approximate.

3. General Purpose Streaming Platforms. General purpose streaming platforms have emerged from demands of defining custom online streaming applications. Query-based systems focus on building efficient plans for certain classes of computation and online algorithmic research focus on providing solutions for specific problems. On the other hand, general purpose frameworks provide platforms for developing and executing streaming applications while automatically supporting application programming, scalability, and fault-tolerance. A natural abstraction of data stream processing is the Graph flow data model, i.e., a DAG of pipelined operators rather than a series of map and reduce stages. The graph contains data-sources that continuously emit data items consumed by the downstream nodes which do the actual processing on the received items. Historically, this has been the core concept in message passing systems which follow a data-driven programming concept. There are two types of abstractions that need to be present: a data source element which emits new data tuples and a data-processing element which defines the logic of data processing.

The current trend towards real-time online processing has fostered a number of open-source and commercial software frameworks for general purpose data stream processing. Next, we describe systems that are specifically designed for general purpose online processing.

Twitter Storm [127] is a distributed stream processing framework that facilitates developing scalable online applications. Storm is a polyglot where it allows writing applications in several languages that are then translated into a logical topology. A topology is a dataflow DAG that represents the required computation. This allows the developer to be only concerned with the computational logic and not about computation distribution. Storm achieves high scalability through horizontal partitioning and uses ZeroMQ [13] for message passing, which ensures low-latency and guaranteed message processing. It offers persistent storage and supports various consistency semantics such as at-least once, at-most once, and exactly-once. On the discovery of a task failure, messages are automatically reassigned by quickly replaying the stream. Squall (Section 3.2) is an analytics engine with a declarative SQL language built on top of Storm. In this thesis, we show how to extend Squall to support general-purpose join processing and skew-resilient operations.

Heron [110] is a next-generation online processing engine developed at Twitter. It is the successor of Storm and is backward compatible with Storm topologies. Heron was built from scratch with the goal of eliminating several performance bottlenecks in Storm. Critical performance issues in Storm arise from layers of indirection during tuple processing. In particular, a worker JVM-process runs several executor threads and each executor is assigned to multiple processing tasks [110]. This design results in non-negligible processing overhead from multiplexing and demultiplexing each tuple through multiple queues and threads within

the different layers. Moreover, multiple levels of indirection cause conflicting scheduling goals and therefore inefficiencies. Heron limits the maximum number of heartbeat-connections using a hierarchical structure of communicating nodes, thereby extending scalability. Heron achieves an order of magnitude performance improvement over Stormⁱⁱ.

Trill [41] and the parallel version, Quill [40], are high performance incremental analytics engines based on the tempo-relational data model. They expose a rich set of data types and user libraries for efficient processing of streaming and relational queries. Additionally, they exploit low level column storage approaches that significantly enhance latency and throughput performance.

Naiad [136] provides a high-level language (LINQ [130]) to support online analytics for cyclic and iterative computation. It does so by presenting a data model that supports global timestamps. In particular, a timestamp signifies the temporal location of a tuple within the dataflow, i.e., location in the DAG, epoch number, and a loop counter. It also supports synchronous and asynchronous computations providing flexibility in developing online applications. Although a global timestamp allows a user to express interesting communication patterns, the scalability and throughput of the system is limited because all tuples need to be timestamped by a centralized entity in the framework.

MillWheel [14] is a framework for low-latency stream processing applications. It exposes a programming model that enables developers to write application logic represented as a custom topology DAG where records are continuously delivered along the edges of the graph. It builds upon efficient fault-tolerant techniques that replay failed tuples, i.e., upstream backup, while eliminating duplicates using Bloom filters.

Amazon Kinesis [18] is a recent commercial web-service that processes real-time massive data from streams. Kinesis allows for custom stream processing as well as query processing. It exposes a Kinesis client library that facilitates applications development using the *producer* and *worker* abstractions. The producer accepts input data from external data streams and processes it to produce a Kinesis stream. The stream consists of data records represented as data tuples which are then consumed by the worker application client. Kinesis automatically adapts and auto-scales to fluctuations in the streaming data rates, providing better resource utilization and lower costs for their customers. It also guarantees fault tolerance by checkpointing and replaying the failed data records.

Flink [20] is an Apache project that got developed from a research project called Stratosphere [16]. This system is designed for online processing and iterative analytics, but it can also support offline processing as a special case of online processing. Flink presents a functional interface that exposes operations, including User defined functions (UDFs), on

ⁱⁱ<http://www.infoq.com/news/2015/06/twitter-storm-heron>

parallel collections. It also has a cost-based optimizer that chooses an optimal query plan with respect to resource utilization, e.g. storage size.

IBM InfoSphere Streams [28] is a commercial high-performance stream processing engine. InfoSphere exposes a declarative programming language, the Stream Processing Language (SPL), to allow developing online applications. SPL allows developers to design applications without worrying about the idiosyncrasies of distributed execution. Users can develop high performant operators using C++ or Java that leverage concurrent processing. A job consists of one or more Processing Elements that communicate using message passing. IBM Streams has a wide domain of commercial applications including transportation, stock market trading, radio astronomy, DNA sequencing, weather forecasting, and telecommunications [28].

2.4 Summary

This chapter presented an introduction to the online stream processing landscape. First we have demonstrated a set of applications that require real-time and interactive computation. Then, we explained the core requirements of efficient stream processing as outlined by Michael Stonebraker *et al.* Finally, we presented the different work that has been done in the field of online processing to demonstrate current applications' demands. We focus on one important requirement for online engines that is to provide an expressive and declarative interface that facilitates analytics on top of streams. This thesis describes techniques, approaches, and the design of systems that support efficient evaluation of online advanced analytics. In particular, this thesis presents approaches that extend SQL-like query computations with efficient general-join processing (chapter 4) and support for a wider range of analytics beyond the relational model. In chapter 5, we present efficient IVM of matrix algebra programs. Matrices can model various domains including machine learning, graph processing, and scientific computing.

3 Squall: Online Query Processing

Analysts process and run exploratory queries on terabytes of data to gain useful insights. Many of these queries include data transformations, e.g., selections and projections, linking and merging with other data sets or streams, e.g., joins, and computing aggregates. This chapter presents an overview of Squall [73], an open-source online distributed query processing engine that allows querying data streams using relational algebra. It exposes several language interfaces that enable SQL-like data manipulation on distributed data streams. Several open-source [20, 127, 110] and commercial [28, 18] systems have been proposed that provide support for SQL-like streaming analytics. However, they do not support arbitrary join processing. This chapter serves as a precursor to the next which presents efficient online theta-joins that extends Squall’s functionality to support efficient general join-processing and skew resilience. We briefly describe the framework and design to pave the road to the next chapter. A detailed discussion about the framework can be found in [175].

3.1 Data Model

Squall’s data model is based on streams of relational tuples. In particular, a stream \mathcal{S} is defined as an unbounded sequence of tuples of the form $\langle \mathfrak{s}, t \rangle$ where \mathfrak{s} is a relational tuple and t is the resulting tuple’s associated timestamp. Relational transformations, e.g., projection and selection, can be applied on stream tuples resulting in another stream of transformed relational tuples.

An un-windowed stream-join between streams \mathcal{S}_1 and \mathcal{S}_2 is defined as the relational-join view between two append-only bags \mathcal{B}_1 and \mathcal{B}_2 . When a new tuple \mathfrak{s}_1 arrives to stream \mathcal{S}_1 , it is added to the corresponding bag, i.e., \mathcal{B}_1 , and then joined against the other bag \mathcal{B}_2 , where the join-results are emitted into a new results-stream \mathcal{S} . New tuples arriving to \mathcal{S}_2 are processed in a symmetrical fashion. There are various semantics for choosing the associated timestamp,

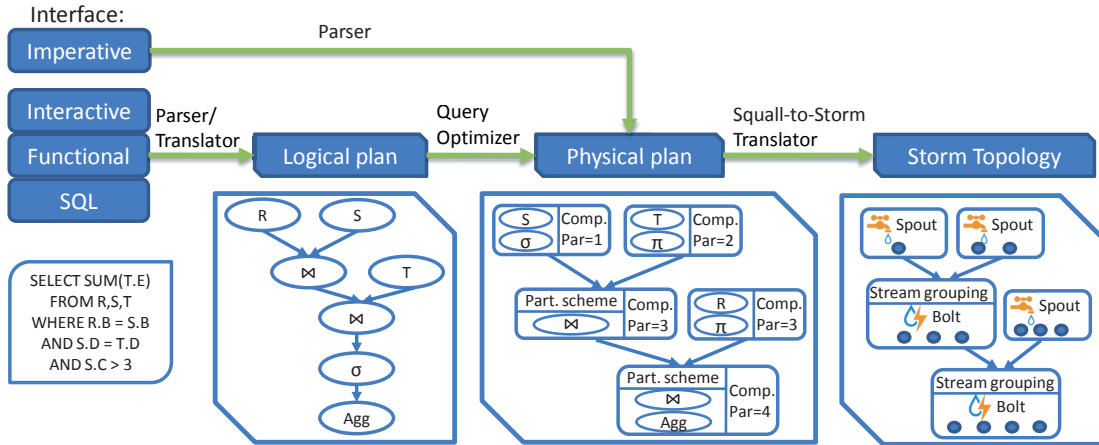


Figure 3.1 – The Squall query processing engine. An example query plan is first translated to a logical plan, then to a physical plan and finally to a storm topology.

e.g., the min or max of the corresponding joined tuples.

Un-windowed streams represent full-history (or landmark-window) semantics. Many other applications have different requirements that do not need to store an unbounded stream of data which might outgrow memory and storage capacity. One possibility is to use window semantics that restrict the scope of computed results over a bounded set of tuples defined within a window. These windows could be defined by temporal or row based semantics [39], generating sliding-window or tumbling-window streams. In the context of joins, a time-based sliding window join of duration t on stream S restricts stream tuples to only join with tuples from the other stream whose timestamp is within the last t time units. A tuple-based sliding window of size k joins with the last k tuples arrived in the stream. Both types of window semantics “slide” forward as time advances or as new tuples arrive. Window semantics enable purging states that has fallen out of the current window as future arrivals cannot possibly join with them anymore.

3.2 The Squall Framework

The stages of running a query within the Squall framework are depicted in Fig. 3.1. In particular, a user writes a query using one of front-end languages which is then translated to a logical plan. A logical plan depicts a high-level algebraic representation of the query. This plan is then optimized and translated to a physical query plan. A physical plan is an annotated plan that contains information about running the query in the physical distributed platform. Finally, the annotated plan is translated to a Storm topology which can be deployed over a Storm cluster. Next, we give an overview of Squall’s main concepts and components.

3.2.1 Interface

Squall offers multiple interfaces for writing streaming queries. We illustrate the different interfaces through the following SQL query from Hyracks [33] that uses the TPCB benchmark. TPCB is a well-known decision support benchmark that demonstrates real world decision support systems on large volumes of data. The interested reader can refer to [6] for more details about the benchmark, schema, and queries. The following query computes the total number of orders placed by customers in various market segments [6] in a five-minutes sliding window.

```
SELECT CUSTOMER.MKTSEGMENT, COUNT(ORDERS.ORDERKEY)
FROM CUSTOMER join ORDERS on CUSTOMER.CUSTKEY=ORDERS.CUSTKEY
GROUP BY CUSTOMER.MKTSEGMENT
RANGE 5 min
```

SQL Interface. Squall exposes a declarative SQL interface that allows writing SQL-like queries on top of streams. Similar to how Hive [168] provides SQL analytics on top of Hadoop [168] for offline processing, Squall provides support for continuous SQL queries on top of Storm. SQL is the de-facto standard for querying relational data. It remains the enduring standard declarative language for databases for four decades. SQL is a widely spread standard that is well-understood by database programmers and is implemented by almost all current DBMS. Therefore, leveraging the familiar SQL querying model to stream processing increases productivity and adoption. In addition, SQL is not only used in relational databases but is also used on top of general purpose distributed systems such as Hadoop. At Facebook, 95% of the Hadoop jobs are generated by Hive, whereas, the remaining 5% are handwritten [151]. Squall supports a wide range of SQL-like queries and constructs that support full-history and window semantics. The previous SQL query is a valid example for continuous-query in Squall.

Scala API. Using functional programming in data analysis has recently gained wide adoption including Spark [188], Flink [20], and Slick [158]. It enables productivity and development convenience. Squall exposes an embedded Scala DSL that allows running different data-transformations on *streams*, e.g., map, flatmap, filter, groupby, etc. The DSL also supports arbitrary compositions, e.g., joining *streams*, using a convenient functional interface. Squall also provides an interactive REPL interface that allows users and developers to interactively construct query plans and run them. The functional (Scala) interface leverages the brevity, productivity, and convenience of functional programming. The previous SQL query is written 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)
val agg = join.groupByKey(x => 1, k => k._1._2).onSlidingWindow(5*60)
agg.execute(conf)
```

Imperative Interface. Finally, Squall provides an imperative interface that gives developers full control in defining exact query plans. This interface allows developers to explicitly add operators into the dataflow chain. For instance, the previous SQL query is imperatively expressed as follows:

```
Component customer = new DataSourceComponent("customer", conf)
    .add(new ProjectOperator(0, 6));
Component orders = new DataSourceComponent("orders", conf)
    .add(new ProjectOperator(1));
Component custOrders = new EquiJoinComponent(customer, 0, orders, 0)
    .add(new AggregateCountOperator(conf)
        .setGroupByColumns(1).setWindowSemantics(5*60));
```

Window Semantics. The previous example demonstrates a sliding window query. Squall supports various flavours of window semantics including sliding, landmark, and tumbling windows. In sliding windows, only some of the window-tuples expire at a given time, whereas tumbling windows evict all tuples in the window at the same time. Landmark windows operate from a fixed time point which could express full-history semantics. In addition to temporal windows, tuple-based windows are also supported. Each line in the following code snippet illustrates a different definition of window semantics on the previous query (using the Scala api):

```
Agg.onWindow(20, 5) // Range 20 secs and slide every 5 seconds
Join.onSlidingWindow(10) // Range 10 seconds and slide every 1 second
Agg.onTumblingWindow(20) // Tumble aggregations every 20 seconds
```

3.2.2 Query plans

Squall translates an input program, written using one of the previous interfaces, to a logical query plan as depicted in Fig. 3.1. A *logical* query plan is an algebraic representation of the query in the form of a DAG of relational algebra operators. After that, after optimization, Squall performs query optimization and generates a *physical* query plan of physical operators which encapsulates additional information related to operator implementation and its corresponding parallelism. In particular, an operator is horizontally distributed across machines with respect to a partitioning scheme. Each partition is assigned to a machine which runs local computations on it. To minimize the amount of shuffled data, consecutive operations that have the same partitioning scheme are co-located as a chain of operations. We refer to these physical operators as *components*. Figure 3.1 demonstrates a physical query plan example with components depicted as rectangular boxes. Notice how query optimization, in this example, pushes selections, projections, and aggregations up the query plan as early as possible to prune unnecessary redundant data. It also co-locates consecutive operations within the same

component to minimize the number of data-routing hops and thus the number of communication messages. Additionally, it re-orders the join operators so as to minimize the size of intermediate results with accordance to the estimated join selectivities. More details about Squall’s query optimization can be found in [175].

3.2.3 Operators

Squall offers a variety of relational operators such as selections, projections, joins, and aggregationsⁱ. Stateless computations, e.g., map operations, can easily scale through horizontal partitioning. However, stateful computations, e.g. joins, are more challenging as they are vulnerable to data-skew and load-imbalance, and are therefore bottlenecks for online processing. An operator consists of a partitioning scheme and a local processing algorithm.

Partitioning schemes

An operator’s partitioning scheme defines the data-route and the state-distribution of input streams across machines. Squall supports *content-sensitive* partitioning schemes like hash-partitioning and range-partitioning. These schemes are useful for computations that require collecting all the relevant data on the same machine to allow correct computations, e.g., hash-partitioning for equi-joins and groupby. These schemes are content-sensitive as they depend on the input’s content to partition the data, e.g., key-hashing. This type of partitioning is vulnerable to data-skew as parallelism and flexibility are limited to potentially coarse-grained keys which might render distributed processing and load balancing infeasible.

Squall also supports *content-insensitive* partitioning schemes [175]. These are schemes that rely on random shuffling and thus, are independent of variances in input data streams. Random shuffling is traditionally used for stateless computations as it enables high scalability. In this thesis, we present *content-insensitive* schemes for stateful join computations that are skew-resilient. The next chapter presents a dataflow operator for efficient arbitrary-join processing.

Local processing

Each machine is assigned a portion of the workload with respect to a specific partitioning scheme. Each node is responsible for processing its assigned portion independently from the other partitions. Local computations have to be non-blocking to prevent hindering online processing. For joins, machines can employ different flavours of non-blocking join algorithms [180, 171, 62, 63, 101, 133]. Squall provides a family of local join algorithms that exploit

ⁱwe currently support SUM, COUNT and AVERAGE aggregates

in-memory indexes to speedup online processing. For instance, equi-joins utilize efficient hash indexes whereas monotonic-joins, e.g., range and band-joins, use balanced binary tree indexes. Indexes are built incrementally and probed on-the-fly. In particular, when a tuple arrives, it is first stored for future processing, where its corresponding index is updated, then it is joined against the indexes of the opposite stream in order to produce the join results. Squall is designed for efficient in-memory processing, however, it offers out-of-core support such as BerkeleyDB [143], which spills tuples to disk whenever memory overflows. However, disk accesses deteriorate latency performance by orders of magnitude. Squall can also support more advanced processing such as incremental view maintenance and approximate online aggregation [175].

3.2.4 Query optimizer

Squall provides a cost-based and rule-based optimizer that automatically creates efficient physical plans. The optimizer tries to find a plan that maximizes throughput and minimizes both latency and resource utilization by choosing a query plan with optimal join order and component-parallelism. Carefully setting component parallelism is important to achieve a balance in producer and consumer queues. This balance prevents from overloading or under-utilizing resources. An overloaded machine suffers from ever-increasing latency and low throughput whereas, under utilized machines waste resources that incur costs especially in cloud environments that employ pay-as-you-go policies. A detailed discussion about Squall's optimizer can be found in [175].

3.2.5 Underlying Processing Platform

Squall uses Twitter's Storm [127] as an underlying distribution platform. However, its design and architecture are applicable to other online general purpose processing engines [175]. Dataflow programs are represented as topologies in Storm. A topology is a DAG of nodes that are horizontally partitioned across physical machines. The nodes can be datasources called spouts or computational node called bolts. A spout emits streams of data items called tuples. Spouts can read data from external sources such as HDFS, Kafka queues, Cassandra, MongoDB, etc and emit the read tuples downstream. On the other hand, bolts consume the emitted tuples to perform general computations. Spouts and bolts are interconnected in the topology graph through stream groupings. A stream grouping represents the routing policy for streaming tuples. Squall is built on top of Storm where it maps a physical query plan to a Storm topology. It is responsible for assigning an efficient implementation of spouts, bolts, and stream groupings that represent the query topology.

3.3 Summary

Squall ⁱⁱ is an online and distributed query processing engine. It is an open-source project that has been designed and developed through collaborative effort. Squall has recently attracted a community of users that rely on it for efficient query processing. It represents an advanced query engine for efficient stream processing. In the next chapter, we present an efficient online join operator that extends Squall to support arbitrary join-predicates and that is resilient to data-skew.

ⁱⁱ<https://github.com/epfldata/squall>

4 Online Theta Joins

A broad range of modern online applications, including fraud-detection mining algorithms, interactive scientific simulations, geosocial network services, and intelligence analysis are characterized as follows: They (i) perform joins (merge) on large volumes of data streams with potentially complex predicates; (ii) require operating in real-time while preserving efficiency and fast response times; (iii) and maintain large state windows, which depend on the history of previously processed tuples [39, 21].

To evaluate joins with generic predicates (known as theta-joins) on very large volumes of data, previous works [161, 141] propose efficient partitioning schemes for *offline* theta-join processing in parallel environments. The goal is to find a scheme that achieves load balancing while minimizing duplicate data storage and network traffic. Offline approaches require that all data is available beforehand and accordingly perform optimization statically before query execution. However, these approaches are not suitable for the online setting. Previous work on stream processing has received considerable attention [7, 21], but is geared towards window-based relational stream models, in which state typically only depends on a recent window of tuples [39]. Although this simplifies the architecture of the stream processing engine, it is ineffective for emerging application demands that require maintaining large historical states [39].

This motivates our work towards efficient theta-join processing in an *online* scalable manner. In this context, the traditional optimize-then-execute strategy is ineffective due to *lack of statistics* such as cardinality information. For pipelined queries, cardinality estimation of intermediate results is challenging because of the possible correlations between predicates [97, 162] and the generality of the join conditions. Moreover, statistics are not known beforehand in streaming scenarios, where data is fed in from remote data sources [61]. Therefore, the online setting requires a versatile dataflow operator that adapts to the data dynamics. Adaptivity ensures low latency, high throughput, and efficient resource utilization throughout the entire

execution.

4.1 Challenges and Contributions

This chapter presents a novel design for an *intra-adaptive* dataflow operator for stateful online join processing. The operator supports arbitrary join-predicates and is resilient to data skew. It encapsulates adaptive state partitioning and dataflow routing. The authors of [82] point out the necessity of investigating systematic adaptive techniques as current ones lack theoretical guarantees about their behavior and instead rely on heuristic-based solutions. Therefore, to design a *provably* efficient operator we need to characterize the optimality measures and the adaptivity costs of the operator. This requires theoretical analysis and addressing several systems design challenges which we discuss while outlining our main contributions.

1. Adapting the partitioning scheme requires state relocation which incurs additional network traffic costs. Our design employs a *locality*-aware migration mechanism that incurs *minimal* state relocation overhead.
2. We present an online algorithm that efficiently decides when to *explore* and *trigger* new partitioning schemes. An aggressively adaptive approach has excessive migration overheads, whereas a conservative approach does not adapt well to data dynamics which results in poor performance and resource utilization. Our presented algorithm balances between maintaining optimal data distribution and adaptation costs. It ensures a constant *competitive ratio* (3.75) in data distribution optimality and *amortized linear* communication cost (including adaptivity costs).
3. Previous adaptive techniques [155, 117, 145] follow a general *blocking*-approach for state relocation that quiesces input streams until relocation ends. Blocking approaches are not suitable for online operators that maintain large states because they incur lengthy stalls. Our design adopts a *non-blocking* protocol for migrations that seamlessly integrates state relocation with *on-the-fly* processing of new tuples while ensuring *eventual consistency* and result correctness.
4. Statistics are crucial for optimizing the partitioning scheme. The operator must gather them on-the-fly and constantly maintain them up-to-date. Traditionally, adaptive solutions delegate this to a centralized entity [155, 117, 85, 182] which may be a bottleneck if the volume of feedback is high [82]. Our approach for computing global statistics is decentralized requiring no communication or synchronization overhead.

Next we discuss related work; Section 4.2 introduces the background and concepts used throughout the rest of the chapter and it outlines the problem and the optimization criteria; Section 4.4 presents the adaptive dataflow operator and its design in detail; and Section 4.5

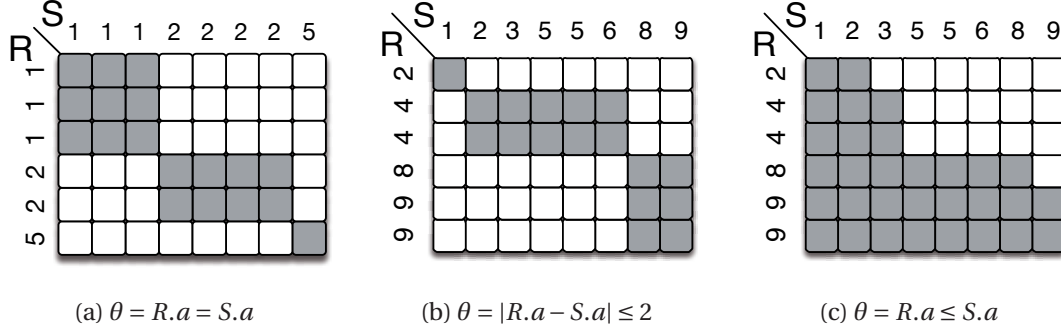


Figure 4.1 – Examples of the join-matrix \mathcal{M} for various (monotonic) joins $R \bowtie_{\theta} S$ between two streams (relations) R and S . Grey cells denote output tuples that satisfy the join predicate θ , alternatively empty cells do not satisfy the predicate.

evaluates performance and validates the presented theoretical guarantees.

4.2 Background & Preliminaries

This section defines notations and conventions used throughout the rest of this chapter. It describes the data partitioning scheme used by the dataflow operator, outlines the operator's structure, and defines the optimization criteria.

4.2.1 Join Partitioning Scheme

We adopt and extend the join-matrix model [141, 161] to the data streaming scenario.

Definition 4.2.1. A join $R \bowtie_{\theta} S$ between two data streams R and S is modeled as a join-matrix \mathcal{M} that represents the cartesian product $R \times S$. For row i and column j , the matrix cell $\mathcal{M}(i, j)$ represents a potential output result. $\mathcal{M}(i, j)$ is true, i.e., an output tuple, if and only if the corresponding tuples r_i and s_j satisfy the join predicate θ . The result of any join is a subset of the cross-product. Hence, the join-matrix model can represent any join condition.

Fig. 4.1 demonstrates a set of join-matrices with monotonic join predicates whereas Fig. 4.2a shows an example of a join-matrix with the predicate \neq .

We assume a shared-nothing architecture where each node operates independently across the cluster. More specifically, none of the nodes share memory or disk storage. J physical machines are dedicated to a *single* join operator. A partitioning scheme maps matrix cells to machines for evaluation such that each cell is assigned to exactly one machine. This ensures result completeness and avoids expensive post processing or duplicate elimination. There are

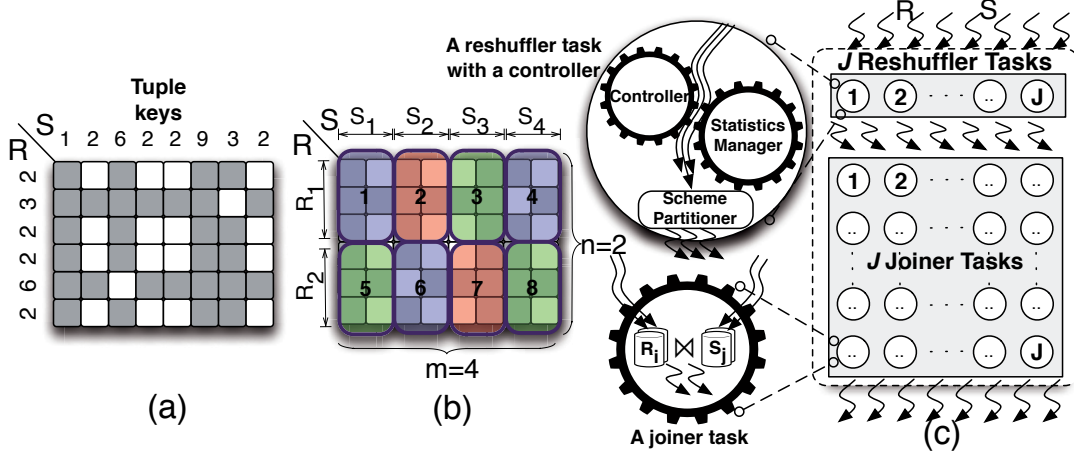


Figure 4.2 – (a) $R \bowtie_{\theta} S$ join-matrix example, grey cells satisfy the $\theta \neq$ predicate. (b) a (2,4)-mapping scheme using $J = 8$ machines. (c) the theta-join operator structure. J reshuffler and J joiner tasks where each physical machine is assigned one from each. One of the reshuffler tasks is designated the additional role of a *controller*.

many possible mappings [141], however, we present a grid-layout partitioning scheme which (i) ensures *minimum* join work distribution among *all* machines, (ii) incurs *minimal* storage and communication costs, (iii) and has a symmetric structure that lends itself to adaptivity. We refer the interested reader to [176] for bounds, proofs, and comparison with previous partitioning approaches [141]. The scheme can be briefly described as follows: to achieve load balance such that each machine is assigned the same number of cells to evaluate, the join-matrix M is divided into J regions of equal area and each machine is assigned a single region. As illustrated in Fig. 4.2b, the streams R and S are split into equally sized stream partitions R_1, R_2, \dots, R_n and S_1, S_2, \dots, S_m where $n \cdot m = J$. For every pair (R_i, S_j) , where $1 \leq i \leq n$ and $1 \leq j \leq m$, there is exactly one machine storing both partitions R_i and S_j . Accordingly, each machine evaluates the corresponding $R_i \bowtie_{\theta} S_j$ independently. We refer to this as the (n, m) -mapping scheme.

4.2.2 Operator Structure

As illustrated in Fig. 4.2c, the operator is composed of two sets of tasks. The first set consists of joiner tasks that do the actual join computation whereas the reshufflers set is responsible for distributing and routing the tuples to the appropriate joiner tasks. An incoming tuple to the operator is randomly routed to a reshuffler task. One task among the reshufflers, referred to as the controller, is assigned the additional responsibility of monitoring global data statistics and triggering adaptivity changes. Each of the J machines run one joiner task and one reshuffler

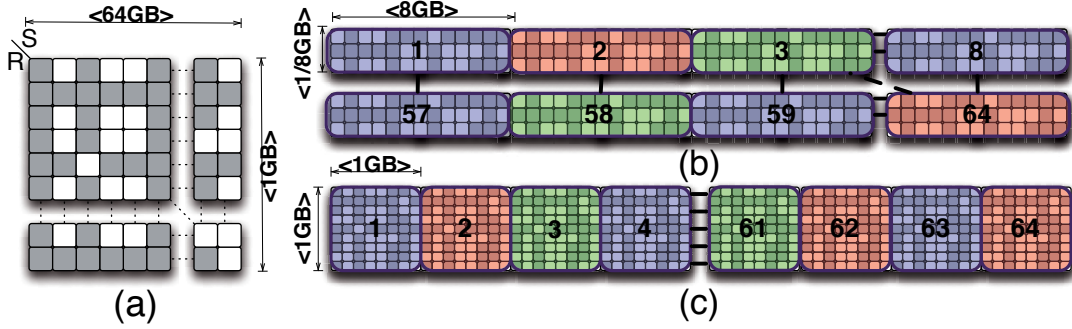


Figure 4.3 – (a) join-matrix with dimensions 1GB and 64GB (b) a $(8, 8)$ -mapping scheme assigns an ILF of $(8\frac{1}{8})$ GB (c) a $(1, 64)$ -mapping scheme assigns an ILF of 2GB.

task.

The reshufflers randomly divide incoming tuples uniformly among stream partitions. Under an (n, m) -mapping scheme, for an incoming r (resp. s) tuple, it is assigned a *randomly* chosen stream partition R_i (resp. S_j). This routing policy ensures load balance and resilience to data skew, i.e., *content-insensitivity*. For a large number of input tuples, the numbers in each partition are roughly equal. Thus, all bounds, later discussed, are meant to approximately hold in expectation with high probability.

Exactly m joiners are assigned partition R_i and exactly n joiners are assigned partition S_j . Therefore, whenever a reshuffler receives a new R (resp. S) tuple and decides that it belongs to partition R_i (resp. S_j), the tuple is forwarded to m (resp. n) distinct joiner tasks. Any flavor of non-blocking join algorithm, e.g., [180, 171, 166, 62, 89], can be independently adopted at each joiner task. Local non-blocking join algorithms traditionally operate as follows: when a joiner task receives a new tuple, it is stored for later use and joined with stored tuples of the opposite relation.

4.2.3 Input-Load Factor (ILF)

Theta-join processing cost, in the presented model, is determined by the costs of joiners receiving input tuples, computing the join, and outputting the result. Under the presented grid-scheme, the join matrix is divided into congruent rectangular regions. Therefore, the costs are the same for every joiner. Since all joiners operate in parallel, we restrict our attention to analyzing one joiner.

The join computation and its output size on a single joiner are independent of the chosen mapping. This holds because both quantities are proportional to the area of a single region,

which is $|R| \cdot |S| / J$. This is independent of n and m . However, the input size corresponds to the **semiperimeter** of one region and is equal to $size_R \cdot |R| / n + size_S \cdot |S| / m$, where $size_R$ ($size_S$) is the size of a tuple of R (S). This also represents the storage required by every joiner since each received tuple is eventually stored. We refer to this value as the **input-load factor** (ILF). This is the only performance metric that depends on the chosen mapping. *An optimal mapping covers the entire join matrix with minimum ILF.* Minimizing the ILF maximizes performance and resource utilization. This is extensively validated in our experiments (Section 4.5) and is attributed to the following reasons: (a) there is a monotonically increasing overhead for processing input tuples per machine. The overhead includes demarshalling the message, appending the tuple to its corresponding storage and index, probing the indexes of the other relation, sorting the input in case of sort-based online join algorithms [62, 133], etc. Minimizing machine input size results in higher local throughput and better performance. (b) Minimizing storage size per machine is also necessary, because performance deteriorates when a machine runs out of main memory and begins to spill to disk. Local non-blocking algorithms perform efficiently when they operate within the memory capacity, however they employ overflow resolution strategies that prevent blocking, but persist to experience performance hits and long delayed join evaluation [61]. (c) Overall, minimizing the ILF results in minimum global duplicate storage and replicated messages ($J \cdot ILF$). This maximizes overall operator performance and increases global resource utilization by minimizing total storage and network traffic and thus preventing congestion. This is essential for cloud infrastructures which typically follow *pay-as-you-go* policies.

Fig. 4.3 compares between two different mappings for a join-matrix with dimensions 1GB and 64GB for streams R and S respectively. Given 64 machines, an $(8, 8)$ -mapping results in an $(8\frac{1}{8})$ GB ILF (region semiperimeter of $8 + \frac{1}{8}$) and a total of 520GB ($8\frac{1}{8} * 64$) of replicated storage and messages. Whereas a $(1, 64)$ -mapping results in a 2GB ILF and a sum of 128GB of replicated data. Since stream sizes are not known in advance, maintaining an optimal (n, m) -mapping throughout execution requires adaptation and mapping changes.

4.2.4 Grid-Layout Partitioning Scheme

The partitioning scheme used throughout the chapter is inspired, but greatly differs from that of [141]. Initially, the number of joiners will be restricted to powers of two. This allows the derivation of bounds (including most notably the input-load factor). Later this assumption will be relaxed. In this subsection, we give some theoretical justification of using this grid-layout scheme with a power of two number of joiners. In the previous work of Okcan *et al.*, the join matrix is divided into square regions with some of the machines left unused. The authors prove that the region semiperimeter and area are within **twice** and **four** times that of the optimal lower bound respectively and are defined as follows:

Theorem 4.2.2. (Okcan *et al.* [141]) Under the mapping scheme discussed in [141], the region semiperimeter is at most $4 \cdot \sqrt{|R||S|/J}$ and the region area is at most $4RS/J$ with the optimal lower bounds being respectively $2 \cdot \sqrt{|R||S|/J}$ and $|R||S|/J$.

Under the grid-layout mapping scheme, allowing rectangular regions rather than restrictive square regions, the bounds derived can be substantially improved.

Theorem 4.2.3. Under the grid-layout mapping scheme, the region semiperimeter is at most 1.07 times the optimal and the region area is exactly $|R||S|/J$ attaining the optimum lower bound.

Proof. The area bound is straightforward. Since there are J regions each with exactly the same area, covering the join matrix, the area is exactly $|R||S|/J$. It remains to show the semiperimeter bound. If the ratio of the relation sizes is J or more, the grid-layout mapping is either $(1, J)$ or $(J, 1)$, being exactly optimal. Otherwise, let the ratio $|R|/|S|$ be ρ where $1/J < \rho < J$. Since n and m are powers of two, it holds that $\frac{1}{2}\rho \leq n/m = n^2/J \leq 2\rho$. The semiperimeter is $|R|/n + |S|/m = \rho|S|/n + |S|n/J$. The maximum value of the semiperimeter is $(\frac{1}{\sqrt{2}} + \sqrt{2})|S|\sqrt{\rho/J} = \sqrt{\frac{9}{8}}|S|\sqrt{\rho/J}$ and is attained at n being either $\sqrt{2\rho J}$ or $\sqrt{\rho J/2}$. This is at most $\sqrt{\frac{9}{8}} = 1.07$ times the optimal lower bound. \square

4.3 Related Work

Parallel Join Processing. In the past decades, much effort has been put into designing distributed and parallel join algorithms to cope with the rapid growth of data sets. Graefe gives an overview of such algorithms in [83]. Schneider *et al.* [154] describe and evaluate several parallel equi-join algorithms that adopt a *symmetric partitioning method* which partitions input on the join attributes, whereas Stamos *et al.* [161] present the *symmetric fragment-and-replicate* method to support parallel theta-joins. This method relies on replicating data to ensure result completeness and on a heuristic model to minimize total communication cost.

MapReduce Joins. MapReduce [56, 5] has emerged as one of the most popular paradigms for parallel computation that facilitates parallel processing of large data and scalability. There has been much work done towards devising efficient join algorithms using this framework. Previous work focuses primarily on equi-join implementations [11, 31, 144, 146, 186] by partitioning the input on the join key, whereas Map-Reduce-Merge [186] supports other join predicates as well. However, the latter requires explicit user knowledge and modifications to the MapReduce model. Recently, Okcan *et al.* [141] proposed techniques that supports theta-join processing without changes to the model. Finally, Zhang *et al.* [191] extend Okcan's work to evaluate multi-way joins.

All of the aforementioned algorithms are offline. They have a blocking behavior that is attributed either to their design or to the nature of the MapReduce framework (the *reduce* phase cannot commence before the *map* phase has completed). In contrast, this thesis sets out to build an online operator that supports scalable processing of theta-joins which allows for early results and rich interactivity.

Online Join Algorithms. There has been great interest in designing non-blocking join algorithms. The symmetric hash join SHJ [180] is one of the first along those lines to support equi-joins. It extends the traditional hash join algorithm to support pipelining. However, the SHJ requires that relations fit in memory. XJOIN [171] and DPHJ [101] extend the SHJ with overflow resolution schemes that allow parts of the hash tables to be spilled out to disk for later processing. Similarly, RPJ [166] uses a statistics-based flushing strategy that tries to keep tuples that are more likely to join in memory. Dittrich *et al.* present PMJ [62, 63] which is a sorting-based online join algorithm that supports inequality predicates as well. Mokbel *et al.* present HMJ [133] that combines the advantages of the two state-of-the-art non-blocking algorithms, namely XJOIN and PMJ. Finally, The family of ripple joins [89] generalize block nested loop join, index loop join, and hash join to their online counterparts. Ripple joins automatically adapt their behavior to provide approximate running aggregates defined within confidence intervals. All the previous algorithms are local online join algorithms, and thus, are orthogonal to our data-flow operator. In the presented parallel operator, each machine can freely adopt any flavor of the aforementioned non-blocking algorithms to perform joins locally on its assigned data partition.

Stream Processing Engines. Distributed stream processors such as BOREALIS [7] and STREAM [21] focus on designing efficient operators for continuous queries. They assume that data streams are processed in several sites, each of which holds some of the operators. They are optimized to handle unbounded streams of data by dropping tuples (load shedding) or having window semantics. In contrast, this thesis is concerned with the design of a scalable operator, as opposed to a centralized approach. And along the same lines of [39], it targets stateful streaming queries which maintain large states, potentially full historical data. Castro *et al.* [39] introduce a scale-out mechanism for stateful operators, however they are limited to stream models with key attributes.

Adaptive Query Processing. Adaptive query processing AQP techniques cope their behavior, at run-time, to data characteristics. There has been a great deal of work on centralized AQP [22, 61, 91, 81] over the last few years. For parallel environments, [82] presents a detailed survey. The FLUX operator [155] is the closest to our work. FLUX is a *general* adaptive operator that encloses adaptive state partitioning and routing. The operator is *content-sensitive* and suitable for look-up based operators. Although the authors focus on single-input aggregate operators [117], it can support a restricted class of join predicates, e.g. equi-join. FLUX

supports equi-joins under skewed data settings but requires explicit user knowledge about partitions before execution. In [85, 178], the authors present techniques to support multi-way non equi-joins. All these approaches are mainly applied to data streaming scenarios with window semantics. On the other hand, this thesis presents an adaptive dataflow operator for general joins. It advances the state of the art in online equi-join processing in the presence of data skew. Most importantly, along the lines of [61, 81, 91], the operator can run on long running full-history queries without window semantics, load shedding, and data arrival order restrictions.

Eddies. Eddies [169, 60] are among the first adaptive techniques known for query processing. Eddies act as a tuple router that is placed at the center of a dataflow, intercepting all incoming and outgoing tuples between operators in the flow. Eddies observe the rates of all the operators and accordingly make decisions about the order at which new tuples will visit the operators. In principal, eddies are able to choose different operator orderings for each tuple within the query processing engine to adapt to the current information about the environment and data. Compared to our work, this direction seeks adaptations at an orthogonal hierarchical level, it is concerned with *inter*-operator adaptivity as opposed to our work on *intra*-operator adaptivity. Moreover, the original eddies architecture is centralized and cannot be applied to a distributed setting in a straightforward manner [82]. However, the work in [169] leverages the eddies design to a distributing setting but assumes window semantics; tolerates loss of information; and neglects adaptations on operators that hold internal state.

4.4 Intra-Operator Adaptivity

We present an intra-operator adaptive approach that modifies its mapping configurations as data flows in. The goal of adaptive processing is, generally, dynamic recalibration to immediately react to the frequent changes in data and statistics. Adaptive solutions supplement regular execution with a control system that monitors performance, explores alternative configurations and triggers changes. These stages are defined within a cycle called the *Adaptivity Loop*. This section presents the design of an adaptive dataflow theta-join operator that continuously modifies its (n, m) -mapping scheme to reflect the optimal data assignment and routing policy. We follow a discussion flow that adopts a common framework [61] that decomposes the adaptivity loop into three stages: (i) The *monitoring* stage that involves measuring data characteristics like cardinalities. (ii) The *analysis and planning* stage that analyzes the performance of the current (n, m) -mapping scheme and explores alternative layouts. (iii) The *actuation* stage that corresponds to migrating from one scheme to another with careful state relocation. In the subsequent discussion, Alg 1 and Alg 2 represent the logic for the first two stages, whereas Alg 4 depict the logic for the final stage. $|R|$, $|S|$ represent the current cardinalities for streams R and S respectively, whereas $|\Delta R|$, $|\Delta S|$ represent the additional (delta) tuples

Algorithm 1 Controller Algorithm.

Input: Tuple t

Initialize: $|R| \leftarrow 0, |S| \leftarrow 0, |\Delta R| \leftarrow 0, |\Delta S| \leftarrow 0$;

```
1: function UPDATE STATE( $t$ )
2:   if  $t \in R$  then
3:      $|\Delta R| \leftarrow |\Delta R| + J$  ▷ Scaled Increment.
4:   else
5:      $|\Delta S| \leftarrow |\Delta S| + J$ 
6:   MigrationDecision( $|R|, |S|, |\Delta R|, |\Delta S|$ )
7:   Route  $t$  according to the current  $(n, m)$ -scheme.
8: end function
```

for R and S , respectively, that have entered the operator.

4.4.1 Monitoring Statistics

In this stage, the operator continuously gathers and maintains online cardinality information of the incoming data. Traditional adaptive techniques in a distributed environment [155, 117, 85, 182] either rely on a centralized controller that periodically gathers statistics or on exchanging statistics among peers [169, 193]. This may become a bottleneck if the number of participating machines and/or the volume of feedback collected is high [82]. In contrast, we follow a decentralized approach, where reshufflers gather statistics on-the-fly while routing the data to joiners. Since reshufflers receive data that is randomly shuffled from the previous stages, the received local samples can be scaled by J to construct global cardinality estimates (Alg 1 lines 3,5). These estimates can be reinforced with statistical estimation theory tools [92] to provide confidence bounds. The advantages of this design are three-fold: *a)* A centralized entity for gathering statistics is no longer required, removing a source of potential bottlenecks. Additionally, it precludes any exchange communication or synchronization overheads. *b)* This model can be easily extended to monitor other data statistics, e.g., frequency histograms. *c)* The design supports fault tolerance and state reconstruction. When a reshuffler or a controller task fails, any other task can take over.

4.4.2 Analysis and Planning

Given that global statistics are constructed in Alg. 1, the controller is capable of analyzing the efficiency of the current mapping scheme, and thus, determining the overall performance of the operator. Furthermore, it checks for alternative (n, m) -mapping schemes that minimize the ILF (Alg 1 line 6). If it finds a better one, it triggers the new scheme. This affects the route of new tuples and impacts machine state. Adopting this dynamic strategy reveals three challenges

that need careful examination: *a)* Since the controller is additionally a reshuffler task, it has the main duty of routing tuples in parallel to exploring alternative mappings. Thus, it has to balance between the ability to quickly react to new cardinality information against the ability to process new tuples rapidly (the classic *exploration-exploitation dilemma*). *b)* Migrating to a new mapping scheme requires careful state maintenance and transfer between machines. This incurs non-negligible overhead due to data transmission over the network. The associated costs of migration might outweigh the benefits if handled naïvely. *c)* An aggressively adaptive control system suffers from excessive migration overheads while a conservative system does not adapt well to data dynamics. Adaptivity thrashing might incur quadratic migration costs. Thus, the controller should be alert in choosing the moments for triggering migrations.

In this section, we describe a constant-competitive algorithm that decides when to *explore* and *trigger* new schemes such that the total cost of communication, including adaptation, is amortized linear.

1.25-Competitive Online Algorithm

Alg. 2 decides the time points that explore and trigger migration decisions. Right after an optimal migration, the system has $|R|$ and $|S|$ tuples from the respective relations. The algorithm maintains two counts $|\Delta R|$ and $|\Delta S|$, denoting the newly arriving tuples on both relations respectively after the last migration. If either $|\Delta R|$ reaches $|R|$ or $|\Delta S|$ reaches $|S|$, the algorithm explores alternative mapping schemes and performs a migration, if necessary.

The two metrics of interest here are the ILF and the migration overhead. The aim of this section is to demonstrate the following key result.

Theorem 4.4.1. *Assume that the number of joiners J is a power of two, the sizes for $|R|$ and $|S|$ are no more than a factor of J apart, and that tuples from R and S have the same size. For a system applying Alg. 2, the following holds:*

1. *The ILF is at most 1.25 times that of the optimal mapping at any point in time. $ILF \leq 1.25 \cdot ILF^*$, where ILF^* is the input-load factor under the optimal mapping. Thus, the algorithm is 1.25-competitive.*
2. *The total communication overhead of migration is amortized, i.e., the time cost of routing a new input tuple, including its migration overhead, is $O(1)$.*

The proof of this theorem is established within the following discussion in this section.

Input-Load Factor. We hereby analyze the behavior of the ILF under the proposed algorithm. Since we assume that $\text{size}(r) = \text{size}(s)$, it follows that minimizing the ILF is equivalent to minimizing $(|R|/n + |S|/m)$.

Chapter 4. Online Theta Joins

Algorithm 2 Migration Decision Algorithm.

Input: $|R|, |S|, |\Delta R|, |\Delta S|$

```

1: function MIGRATIONDECISION( $|R|, |S|, |\Delta R|, |\Delta S|$ )
2:   if  $|\Delta R| \geq |R|$  or  $|\Delta S| \geq |S|$  then
3:     Choose mapping  $(n, m)$  minimizing  $\frac{|R|+|\Delta R|}{n} + \frac{|S|+|\Delta S|}{m}$ 
4:     Decide a migration to  $(n, m)$ 
5:      $|R| \leftarrow |R| + |\Delta R|; |S| \leftarrow |S| + |\Delta S|$ 
6:      $|\Delta R| \leftarrow 0; |\Delta S| \leftarrow 0$ 
7:   end function

```

Lemma 4.4.2. *If J is a power of two and it holds that $1/J \leq |R|/|S| \leq J$, then under an optimal mapping (n, m) ,*

$$\frac{1}{2} \frac{|S|}{m} \leq \frac{|R|}{n} \leq 2 \frac{|S|}{m} \quad \text{and} \quad \frac{1}{2} \frac{|R|}{n} \leq \frac{|S|}{m} \leq 2 \frac{|R|}{n}.$$

Proof. An optimal mapping minimizes $|R|/n + |S|/m$, under the restriction that $n \cdot m = J$. This happens when $|R|/n$ and $|S|/m$ are closest to each other. Since J is a power of two, by assumption, (and also n and m), it follows that under the optimal mapping $|R|/n \leq 2|S|/m$. Assume it were not the case, then $|R|/n > 2|S|/m$. Under the mapping $(2n, m/2)$, both $|R|/n$ and $|S|/m$ are closer, yielding a lower input-load factor, contradicting the optimality of (n, m) . Choosing such a mapping is possible, assuming that $1/J \leq |R|/|S| \leq J$. The other inequality is symmetric. \square

This lemma is useful in proving all subsequent results. The first important result is that the ILF is within a constant factor from that of the optimal scheme. This is due to the fact that Alg. 2 does not allow the operator to receive many tuples without deciding to recalibrate. The following theorem formalizes this intuition.

Lemma 4.4.3. *If $|\Delta R| \leq |R|$ and $|\Delta S| \leq |S|$ and (n, m) is the optimal mapping for $(|R|, |S|)$ tuples, then the optimal mapping for $(|R| + |\Delta R|, |S| + |\Delta S|)$ is one of (n, m) , $(n/2, 2m)$, and $(2n, m/2)$.*

Proof. Without loss of generality, assume that $|\Delta S| \geq |\Delta R|$. It holds that an optimal mapping will not decrease m (since $|S|$ grew relative to $|R|$). Therefore, the optimal is one of (n, m) , $(n/2, 2m)$, $(n/4, 4m)$, \dots , etc. To prove that the optimum is either (n, m) or $(n/2, 2m)$, it is sufficient to prove the following inequality

$$\begin{aligned} \frac{|R|+|\Delta R|}{n/2} + \frac{|S|+|\Delta S|}{2m} &\leq \frac{|R|+|\Delta R|}{n/4} + \frac{|S|+|\Delta S|}{4m} \\ \frac{|S|+|\Delta S|}{m} &\leq \frac{8(|R|+|\Delta R|)}{n} \end{aligned}$$

which means that the ILF under an $(n/2, 2m)$ -mapping is smaller than that under an $(n/4, 4m)$ -mapping. This holds because $|S|/m \leq 2|R|/n$ (lemma 4.4.2), even if $|\Delta S| = |S|$ and $|\Delta R| = 0$. The case $|\Delta R| \geq |\Delta S|$ is symmetric. \square

Alg. 2 decides migration once $|\Delta R| = |R|$ or $|\Delta S| = |S|$. Therefore, lemma 4.4.3 implies that while the system is operating with the mapping (n, m) , the optimum is one of (n, m) , $(n/2, 2m)$, and $(2n, m/2)$. This implies the following.

Lemma 4.4.4. *If $|\Delta R| \leq |R|$ and $|\Delta S| \leq |S|$ and (n, m) is the optimal mapping for $(|R|, |S|)$ tuples, then under Alg. 2, the input-load factor ILF never exceeds $1.25 \cdot \text{ILF}^*$. In other words, the algorithm is 1.25-competitive.*

Proof. By lemma 4.4.3, the optimal mapping is either (n, m) , $(n/2, 2m)$ or $(2n, m/2)$. If the optimal mapping is (n, m) then $\text{ILF} = \text{ILF}^*$. Otherwise, the ratio can be bounded as follows. Without loss of generality, assume that the optimum is $(n/2, 2m)$ then

$$\frac{\text{ILF}}{\text{ILF}^*} \leq \frac{(|R| + |\Delta R|)/n + (|S| + |\Delta S|)/m}{(|R| + |\Delta R|)/(n/2) + (|S| + |\Delta S|)/(2m)}$$

where the constraints $|\Delta R|/n \leq |R|/n$, $|\Delta S|/m \leq |S|/m$ and those in lemma 4.4.2 must hold. All cardinalities are non-negative. Consider the ratio as a function of the variables $|R|/n$, $|S|/m$, $|\Delta R|/n$ and $|\Delta S|/m$. The maximum value of the ratio of linear functions in a simplex (defined by the linear constraints) is attained at a simplex vertex. By exhaustion, the maximum occurs when $|\Delta R| = 0$, $|\Delta S| = |S|$ and $|S|/m = 2|R|/n$. Substituting gives 1.25. \square

Migration Overhead. It remains to show that, under the described algorithm, the migration overhead is amortized. This requires showing that the migration process can be done efficiently and that when a migration is triggered, enough tuples are received to “pay” for this migration cost.

The migration of interest is the change from the (n, m) to $(n/2, 2m)$ -mapping (symmetrically, (n, m) to $(2n, m/2)$). Migration can be done naïvely by repartitioning all previous states around the joiners according to the new scheme. This approach unnecessarily congests the network and is expensive. In contrast, we present a *locality*-aware migration mechanism that minimizes state transfer overhead. To illustrate the procedure, we walk through an example. Consider a migration from a $(8, 2)$ to a $(4, 4)$ -mapping scheme ($J = 16$) as depicted in Fig. 4.4. Before the migration, each joiner stores about an eighth of R and half of S . After the migration, each joiner stores a quarter of R and only one quarter of S . To adapt, joiners can efficiently and deterministically discard a quarter of S (half of what they store). However, tuples of R must be exchanged. In Fig. 4.4, joiners 1 and 2 store the “first” eighth of R while joiners 3 and 4 store

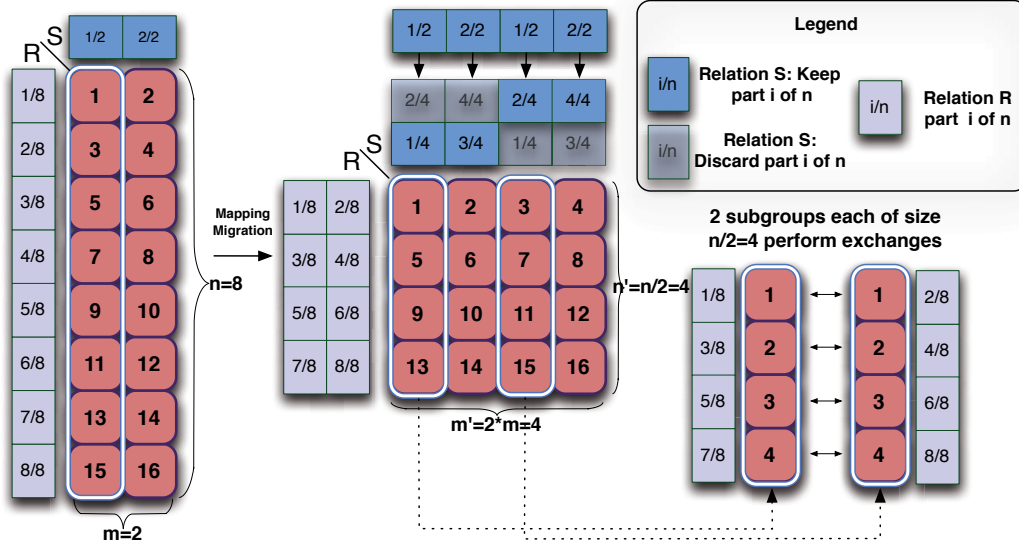


Figure 4.4 – Migration from a (8,2)- to a (4,4)-mapping. Discards are performed on the state of stream S and exchanges are performed on the state of stream R.

the “second” eighth of R . Joiners 1 and 3 can exchange their tuples and joiners 2 and 4 can do the same in parallel. Joiners 5 and 7, 6 and 8, and so forth operate similarly in parallel. This incurs a total overhead of $|R|/4$ time units which is the bi-directional communication cost of $|R|/8$. This idea can be generalized, yielding bounds on the migration overhead.

Lemma 4.4.5. *Migration from (n, m) to $(n/2, 2m)$ -mapping can be done with a communication cost of $2|R|/n$ time units. Similarly, migrating to $(2n, m/2)$ incurs a cost of $2|S|/m$.*

Proof. Without loss of generality, consider the migration to $(n/2, 2m)$. No exchange of S state is necessary. On the other hand, tuples of R have to be exchanged among joiners. Before migration each of the J joiners had $|R|/n$ tuples from R , while after the migration, each must have $2|R|/n$. Consider one group of n joiners sharing the same tuples from S (corresponding to a “column” in Fig. 4.4). These joiners, collectively, contain the entire state of R . They can communicate in parallel with the other $m-1$ groups. Therefore, we analyze the state relocation for one such group and it follows that all groups behave similarly in parallel.

Divide the group into two subgroups of $n/2$ joiners. Number the joiners in each group $1, 2, \dots, n/2$. Joiner pairs labeled i should exchange their tuples together. It is clear that each pair of joiners labeled i ends up with a distinct set of $2|R|/n$ tuples. Fig. 4.4 describes this exchange process. Each of the pairs labeled i can communicate completely in parallel. Therefore, the total migration overhead is $2|R|/n$, since each joiner in the pair sends $|R|/n$

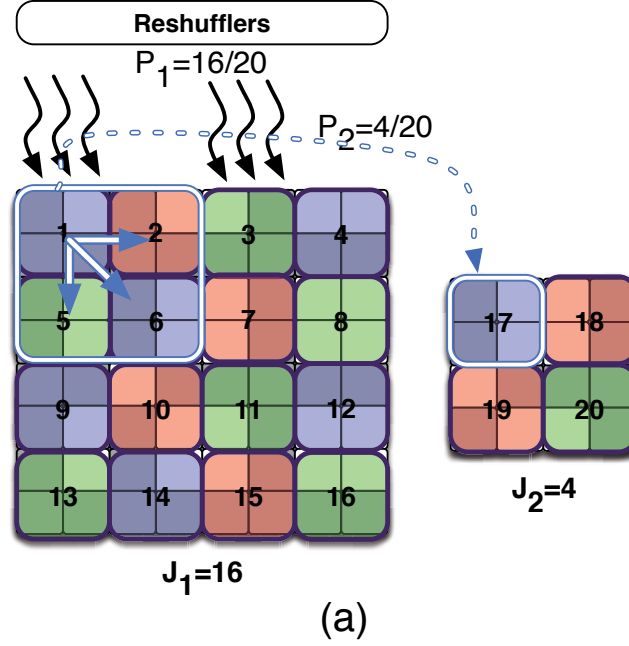


Figure 4.5 – (a) decomposing $J = 20$ machines into independent groups of 16 and 4 machines. Tuple storage within a group is defined by the probability measures.

tuples to the other. □

Lemma 4.4.6. *The cost of routing tuples and data migration is linear. The amortized cost of an input tuple is $O(1)$.*

Proof. Since all joiners are symmetrical and operate simultaneously in parallel, it suffices to analyze cost at one joiner. Therefore, after receiving $|\Delta R|$ and $|\Delta S|$ tuples, the operator spends *at least* $\max(|\Delta R|/n, |\Delta S|/m)$ units of time processing these tuples at the appropriate joiners. By assigning a sufficient amortized cost per time unit, the received tuples pay for the later migration.

By lemma 4.4.3, the optimal mapping is (n, m) , $(n/2, 2m)$ or $(2n, m/2)$. If the optimal mapping is (n, m) , then there is no migration. Without loss of generality, assume that $|\Delta S| \geq |\Delta R|$ and that the optimal mapping is $(n/2, 2m)$. Between migrations, $\max(|\Delta R|/n, |\Delta S|/m)$ time units elapse, each is charged 7 units. One unit is used to pay for routing and 6 are reserved for the next migration. The cost of migration by lemma 4.4.5 is $2(|R| + |\Delta R|)/n$. The amortized cost reserved for migration is $6 \max(|\Delta R|/n, |\Delta S|/m)$. Since a migration was triggered, either $|\Delta R| = |R|$ or $|\Delta S| = |S|$. In either case, it should hold that the reserved cost is at least the

migration cost, that is,

$$6 \max(|\Delta R|/n, |\Delta S|/m) \geq 2(|R| + |\Delta R|)/n.$$

If $|\Delta R| = R$, then by substituting, the left hand side is $6 \max(|\Delta R|/n, |\Delta S|/m) \geq 6|R|/n$ and the right hand side is $2(|R| + |\Delta R|)/n = 4|R|/n$. Therefore, the inequality holds. If $|\Delta S| = S$, then the left hand side is

$$6 \max(|\Delta R|/n, |\Delta S|/m) \geq 2|\Delta R|/n + 4|S|/m.$$

Therefore, the left hand side is not smaller than the right hand side, since $2|S|/m \geq |R|/n$ (by lemma 4.4.2). Thus, the inequality holds in both cases. The cases, when $|\Delta R| \geq |\Delta S|$ or when the optimal is $(2n, m/2)$, are symmetric. \square

Lemmas 4.4.4 and 4.4.6 directly imply Theorem 4.4.1.

Generalization and Discussion

In the previous section, the analysis was based upon three assumptions: the cardinality ratio of the larger relation to the smaller relation does not exceed J ; the number of joiners is a power of two; and tuples from R and S have the same size. In this section we outline how to relax these assumptions and show that the algorithm remains to have a constant-competitive ratio and the migration overhead cost persists to be amortized and linear in the number of input tuples.

Relation Cardinality Ratio. Without loss of generality, assume that $|R| > |S|$. The analysis in the previous section assumed that $|R| \leq J|S|$. This can be relaxed by continuously padding the smaller relation with dummy tuples to maintain the ratio less than J . This requires padding the relation S with at most $|R|/J \leq T/J$ tuples, where T is the total number of tuples $|R| + |S|$. Therefore, the total number of tuples the operator handles, including dummy tuples, is at most $T + T/J = (1 + 1/J)T$ tuples. The ratio of the relation sizes still respects the assumption. Therefore, the analysis in the previous section holds except that the ILF now gets multiplied by a factor of $1 + 1/J$. This factor is at most 1.5 (since $J \geq 2$). This factor *tends to one* as the number of joiners increases. Therefore, the algorithm is still constant-competitive, with the constant being $1.25 \cdot 1.5 = 1.875$. Similarly, adding the dummy tuples multiplies the migration overhead by at most 1.5. Therefore, the communication overhead remains linear.

Number of Joiners. Assume that $J \geq 1$, then J has a unique decomposition into a sum of powers of two. Let $J = J_1 + J_2 + \dots + J_c$ where each J_i is a power of two. Accordingly, the machines are broken down into c groups, where group i has J_i machines. There can be at most $\lceil \log J \rceil$ of such groups. Finally, each group operates exactly as described in the previous section. Fig. 4.5a illustrates an example, given a pool of $J = 20$ machines, it is clustered into

three groups of sizes 16 and 4 which operate independently. An incoming tuple is sent to all c groups to be joined with all stored tuples. Only one group stores this tuple for joining with future tuples. The group that stores this tuple is determined by computing a pseudo-random hash whose ranges are proportional to the group sizes. The probability that group i is chosen is equal to $P_i = J_i / J$. With high probability, after T tuples have been received, the number of tuples stored in group i is close to $(J_i / J)T$.

It is essential that if a pair of tuples is sent to two machines, each belonging to a different group, that this pair of tuples is received in the same order by both machines. With very high probability (after a small number of tuples has been received), the mappings of two groups will be similar. More specifically, for two groups with sizes $J_1 < J_2$, it will hold that n_2 (m_2) is divisible by n_1 (m_1). Blocks of machines in the bigger group correspond to a single machine in the smaller group (see figure 4.5). In each such block, a single machine does the task of forwarding all tuples to machines within that block as well as the machine in the smaller group (see the same figure). This ensures that machines get tuples in the same order at the cost of tuple latency proportional to $\log J$, since tuples have to be propagated serially among $\log J$ groups of machines.

Let the biggest group be L with size J' which is at least half of J . The storage is bounded by that of L (receiving the entire input). The optimal storage is at most half that of L (since J' is at least half of J). Therefore, the competitive ratio of storage is at most doubled ($1.875 * 2 = 3.75$). Since groups operate independently, migrations are performed asynchronously and completely in parallel. Therefore, only tuple routing gets multiplied by a $\log J$ factor, since every tuple is broadcast to at most $\log J$ groups. Therefore, the total routing cost, including migrations, is $O(T \log J)$.

It remains to show that the described distribution of data does not affect the original configuration that all joiners perform an equal amount of join work. Without loss of generality, consider two tuples t_R and t_S where t_R arrives to the system before t_S (the other case is symmetric). We show that the probability a specific joiner j computes $\{t_R\} \bowtie \{t_S\}$ is $1/J$, implying directly that the work gets equally distributed. For joiner j to perform the join, t_R has to be stored on j . The probability of this happening is $(J_g / J) \cdot (1/n_g)$ where J_g is the group size of group g containing joiner j and n_g is the number of rows in the mapping of this group. t_S gets communicated to all groups. The probability that t_S is sent to j is exactly $1/m_g$ where m_g is the number of columns in the mapping of group g . Multiplying both probabilities and noticing that $n_g \cdot m_g = J_g$ gives exactly $1/J$.

Optimality-Communication Tradeoff. It is possible to modify Alg. 2 to tradeoff the mapping optimality with the communication overhead. The algorithm checks for the possibility of performing migration whenever either $|\Delta R| = |R|$ or $|\Delta S| = |S|$. By modifying these conditions to be $|\Delta R| = \epsilon |R|$ or $|\Delta S| = \epsilon |S|$, where $0 < \epsilon \leq 1$, we directly get a tradeoff between optimality

and communication cost.

Theorem 4.4.2. *Under the modified Alg. 2 (parameterized by ϵ), the competitive ratio of the ILF becomes $\frac{3+2\epsilon}{3+\epsilon}$ and the amortized communication cost becomes $\frac{7}{\epsilon} = O(\frac{1}{\epsilon})$.*

Proof. The proof is exactly following the lemmas of subsection 4.4.2 and replacing the conditions $|\Delta R| \leq |R|$ and $|\Delta S| \leq |S|$ by $|\Delta R| \leq \epsilon |R|$ and $|\Delta S| \leq \epsilon |S|$, respectively. The competitive ratio is given by the following expression:

$$\frac{ILF}{ILF^*} \leq \frac{(|R| + |\Delta R|)/n + (|S| + |\Delta S|)/m}{(|R| + |\Delta R|)/(n/2) + (|S| + |\Delta S|)/(2m)}$$

This attains its maximum value $\frac{3+2\epsilon}{3+\epsilon}$ at $|\Delta R| = 0$, $|\Delta S| = \epsilon |S|$ and $|S|/m = 2|R|/n$.

For every input tuple, an amortized cost of $3 + 4/\epsilon$ is given. Between migrations, at least $\max(|\Delta R|/n, |\Delta S|/m)$ are received. Without loss of generality, the migration cost is $2 \frac{|R|+|\Delta R|}{n}$. If $|\Delta R| = \epsilon |R|$, substituting shows that the amortized cost exceeds the migration cost. In the case of $|\Delta S| = \epsilon |S|$, substituting and noting that $\frac{S}{m} \geq \frac{R}{2m}$ (by lemma 4.4.2), it also holds that the total migration cost is less than the amortized cost. The theorem statement immediately follows. \square

Notice that by setting $\epsilon = 1$, Theorem 4.4.1 is recovered.

Elasticity. In the context of online query processing, the query planner may be unable to a-priori determine the number of machines J to be dedicated to a join operator. It is thus desirable to allocate as few joiners as possible to the operator while ensuring that the stored state on each machine is reasonably maintained to prevent disk spills and performance degradation. We hereby present a scheme that allows the join operator to elastically expand using more machines, as needed, while maintaining all the theoretical bounds described (merely constant changes in the communication cost).

For joiners, designate a maximum number M of tuples (ILF) per joiner. At migration checkpoints (following theorem 4.4.2 when $|\Delta R| = \epsilon |R|$ or $|\Delta S| = \epsilon |S|$), after migration, if each joiner stores a number of tuples exceeding $M/2$, the system expands by splitting every joiner into 4 joiners. Every joiner communicates its tuples to three new joiners as described in Fig. 4.6. This can be done with a total communication cost equal to twice the number of tuples stored on that joiner prior to expansion.

Under this scheme, it is obvious that the competitive ratio of the ILF is unaffected, since splitting every machine to four machines does not change the ratio of n to m . It remains to show that the amortized cost of communication is not much affected.

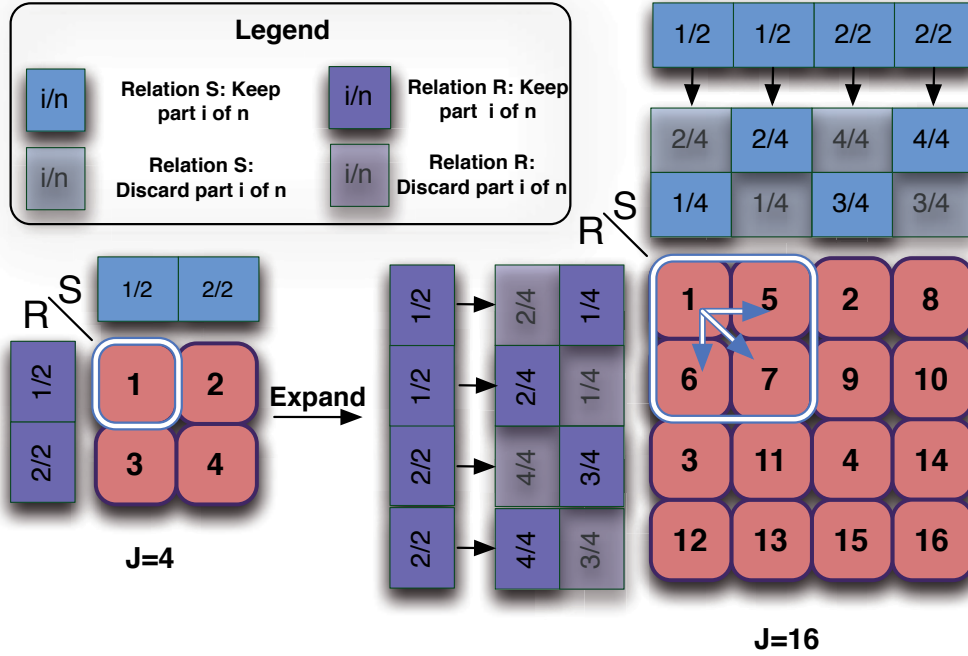


Figure 4.6 – This figure illustrates the expansion of the system. Each machine state is distributed to 4 joiners. Each joiner communicates the appropriate portions of its state to the three new joiners. For example, joiner 1 sends the second half of its S tuples to joiners 5 and 7. It sends the second half of its R tuples to 6 and 7. It also sends the first half of R to 5 and the first half of S to 6.

Theorem 4.4.3. *Under the modified Alg. 2 (parameterized by ϵ), the described expansion has an amortized cost of $\frac{8}{\epsilon} = O(1/\epsilon)$.*

Proof. After receiving $|\Delta R|$ and $|\Delta S|$ tuples, the operator spends at least $\max(|\Delta R|/n, |\Delta S|/m)$ units of time processing these tuples at the appropriate joiners. Each is assigned an amortized cost of $4 + 4/\epsilon \leq 8/\epsilon$. The communication cost due to expansion is at most $2(\frac{|R|+|\Delta R|}{n} + \frac{|S|+|\Delta S|}{m})$. $4\max(|\Delta R|/n, |\Delta S|/m)$ is used to account for $2|\Delta R|/n + |\Delta S|/m$. It remains to notice that $\frac{4}{\epsilon}\max(|\Delta R|/n, |\Delta S|/m) \geq 2(|R|/n + |S|/m)$ since either $|\Delta R| = \epsilon|R|$ or $|\Delta S| = \epsilon|S|$ and since $\frac{1}{2}\frac{|R|}{n} \leq \frac{|S|}{m} \leq 2\frac{|R|}{n}$ (by lemma 4.4.2). \square

Relative Tuple Sizes. Let the sizes of an R tuple and an S tuple be $\text{size}(r) = \tau_R$ and $\text{size}(s) = \tau_S$ respectively. An input R tuple can be viewed as the reception of τ_R “unit” tuples. Similarly an S tuple is τ_S unit tuples. The previous analysis holds except that migration decisions

can be slightly delayed. For example, if the migration decision is supposed to happen after the reception of 5 unit tuples and a tuple of size 1000 units is received, then the migration decision is delayed by 995 units. Therefore, the ILF is increased by at most an additive factor of $\max(\tau_R, \tau_S)$, i.e., $ILF \leq K \cdot ILF^* + \max(\tau_R, \tau_S)$.

Window Semantics. Until now, we have discussed an *append-only* state model. This is common in systems that compute online approximations of query results such as those adopted in online aggregation systems [89, 92, 102]. Under this setting, input data from static databases are continuously fed into the running query to produce early approximate results. Furthermore, these results can be defined within confidence bounds using statistical estimation theory tools. In this case state continuously grow as more input data is fed in. The *temporal* state model represents a more dynamic setting which allows insertions and state-purge. This is adopted in the more common window-semantics query processing engines [155, 7, 21]. Our work can be extended to support the temporal state model, and therefore window semantics. The migration decision algorithm and the analysis are slightly tweaked to take into consideration tuple-purge along with insertions.

Adaptation Loop Revisited. Alg. 3 decides the time points at which adaptation decisions are triggered. Right after an optimal migration, the operator has received $|R|$ and $|S|$ tuples from the respective relations. The algorithm maintains two counts $|\Delta R|$ and $|\Delta S|$, representing the newly **arrived and purged** tuples on both relations respectively after the last migration. If either $|\Delta R|$ reaches $|R|/2$ or $|\Delta S|$ reaches $|S|/2$, the algorithm explores alternative mapping schemes and performs a migration, if necessary.

The two metrics of interest here are the ILF and the migration overhead.

Theorem 4.4.4. *Assume that the number of joiners J is a power of two, the sizes for $|R|$ and $|S|$ are no more than a factor of J apart, and that tuples from R and S have the same size. An adaptive scheme that applies Alg. 3 ensures the following characteristics:*

1. *The ILF is at most 1.4 times that of the optimal mapping at any point in time. $ILF \leq 1.4 \cdot ILF^*$, where ILF^* is the input-load factor under the optimal mapping at any point in time. Thus, the algorithm is 1.4-competitive.*
2. *The total communication overhead of migration is amortized, i.e., the cost of routing a new input tuple, including its migration overhead, is $O(1)$.*

Proofs and analyses are illustrated in the appendix.

Algorithm 3 Migration Decision Algorithm (with deletions).**Input:** $|R|, |S|, |\Delta R|, |\Delta S|$

```

1: function MIGRATIONDECISION( $|R|, |S|, |\Delta R|, |\Delta S|$ )
2:   if  $|\Delta R| \geq |R|/2$  or  $|\Delta S| \geq |S|/2$  then
3:     Choose mapping  $(n, m)$  minimizing  $\frac{|R|+|\Delta R|}{n} + \frac{|S|+|\Delta S|}{m}$ 
4:     Decide a migration to  $(n, m)$ 
5:      $|R| \leftarrow |R| + |\Delta R|; |S| \leftarrow |S| + |\Delta S|$ 
6:      $|\Delta R| \leftarrow 0; |\Delta S| \leftarrow 0$ 
7: end function

```

4.4.3 Actuation

The previous section provides a high-level conceptual description of the algorithm. Migration decision points are specified to guarantee a close-to-optimal ILF and linear amortized adaptivity cost. This section describes the system-level implementation of the migration process.

Previous work on designing adaptive operators [155, 117, 145] follow a general theme for state relocation. The following steps give a brief description of the process: (i) Stall the input to the machines that contain state to be repartitioned. The new input tuples are buffered at the data sources. (ii) Machines wait for all in-flight tuples to arrive and be processed. (iii) Relocate state. (iv) Finally, online processing resumes. Buffered tuples are redirected to their new location to be processed. This protocol is not suitable for stateful operators. Its blocking behavior causes lengthy stalls during online processing until state relocation ends.

Eventually Consistent Protocol

It is essential for the operator to continue processing tuples *on-the-fly* while performing adaptations. Achieving this presents new challenges to the correctness of the results. When the operator migrates from one partitioning scheme \mathcal{M}_i to another \mathcal{M}_{i+1} it undergoes a state relocation process. During this, the state of each machine, within the operator, does not represent a state that is consistent with either \mathcal{M}_i or \mathcal{M}_{i+1} . Hence, it becomes hard to reason about how new tuples entering the system should be joined. This section presents a non-blocking protocol that allows continuous processing of new tuples during state relocation by reasoning about the state of any tuple circulating the system with the help of *epochs*. This ensures that the system (i) is consistent at all times except during migration, (ii) eventually converges to the consistent target state \mathcal{M}_{i+1} , and (iii) produces correct and complete join results in a continuous manner.

The operation of the system is divided into *epochs*. Initially, the system is in epoch zero.

Whenever the controller decides a mapping change, the system enters a new epoch with incremented index. For example, if the system starts with the mapping (8, 8), later migrates to (16, 4) and finally migrates to (32, 2), the system went through exactly three epochs. All tuples arriving between the first and the second migration decision belong to epoch 1. All tuples arriving after the last mapping-change decision belong to epoch 2. Reshufflers and joiners are not instantaneously aware of the epoch change, but continue to process tuples normally until they receive an epoch change signal along with the new mapping. Whenever a reshuffler routes a tuple to joiners, it tags it with the latest epoch number it is aware of. It is crucial for the correctness of the scheme described shortly to guarantee that all machines are at most one epoch behind the controller. That is, all machines operate on, at most, two different epochs. This is, however, guaranteed theoretically and formalized later in Theorem 4.4.7.

The migration starts by the controller making the decision. The controller broadcasts to all reshufflers the mapping change signal. When a reshuffler receives this signal, it notifies all joiners and immediately starts sending tuples in accordance to the new mapping. Joiners continuously join incoming tuples and start exchanging migration tuples. Once a joiner has received epoch change signals from *all* reshufflers, it is guaranteed that it will receive no further tuples tagged with the old epoch index. At that point, the joiner proceeds to finalize the migration and notifies the controller once it is done. The controller can only start a new migration once all joiners notify it that they finished the data migration. The subsequent discussion shows how joiners continue processing tuples while guaranteeing consistent state and correct output.

The timestamp of the migration decision at the controller partitions the tuples into several sets. During a migration, τ is the set of all tuples received before the migration decision. μ is the set of all tuples that are sent from one joiner to another (due to migration). The set of new tuples received after the migration decision timestamp are either tagged with the old epoch index, referred to as Δ , or with the new epoch index, referred to as Δ' . Notice that $\mu \subset (\tau \cup \Delta)$. To simplify notation, no distinction is made between tuples of R or S . For example, writing $\Delta \bowtie \Delta'$ refers to $(\Delta_R \bowtie \Delta'_S) \cup (\Delta_S \bowtie \Delta'_R)$, where σ_R (σ_S) refers to the tuples of R (S) in the set σ .

During the migration, joiners have tuples tagged with the old epoch and the new epoch. Those tuples tagged with the new epoch are already on the correct machines since the reshuffler sent them according to the new mapping. Joiners should redistribute the tuples tagged with old labels according to the new mapping. The set of tuples tagged with the old label is exactly $\tau \cup \Delta$. Joiners discard portions and communicate other portions to the other machines. The discarded tuples are referred to as $\text{DISCARD}(\tau \cup \Delta)$. For convenience, $(\tau \cup \Delta) - \text{DISCARD}(\tau \cup \Delta)$ is referred to as $\text{KEEP}(\tau \cup \Delta)$. The migrated tuples are $\text{MIGRATED}(\tau \cup \Delta)$ which coincides exactly with μ . $\text{KEEP}(\tau)$ refers to tuples in $\text{KEEP}(\tau \cup \Delta) \cap \tau$. The same holds for DISCARD , MIGRATED

and the set Δ .

Definition 4.4.5. *A migration algorithm is said to be correct if right after the completion of a migration, the output of the system is exactly $(\tau \cup \Delta \cup \Delta') \bowtie (\tau \cup \Delta \cup \Delta')$.*

During the migration, the output may be incomplete. Therefore, completeness and consistency are defined only upon the completion of the migration. The complete output is the join of all tuples that arrived to the system before (τ) and after the migration decision $(\Delta \cup \Delta')$. Alg. 4 describes the joiner algorithm. The output of the algorithm is provably correct. For the proof of correctness, an alternative characterization of the correct output is needed.

Lemma 4.4.7.

$$(\tau \cup \Delta \cup \Delta') \bowtie (\tau \cup \Delta \cup \Delta')$$

is equivalent to the union of (1) $\tau \bowtie \tau$, (2) $\Delta \bowtie \Delta$, (3) $\tau \bowtie \Delta$, (4) $\Delta' \bowtie \mu$, (5) $\Delta' \bowtie \text{KEEP}(\Delta)$, (6) $\Delta' \bowtie \text{KEEP}(\tau)$, and (7) $\Delta' \bowtie \Delta'$.

Proof. Since set union distributes over join, the result can be rewritten as

$$(\tau \bowtie \tau) \cup (\tau \bowtie \Delta) \cup (\tau \bowtie \Delta') \cup (\Delta \bowtie \Delta) \cup (\Delta \bowtie \Delta') \cup (\Delta' \bowtie \Delta').$$

The subsets (1), (2), (3) and (7) appear directly in the expression. It remains to argue that $\Delta' \bowtie (\tau \cup \Delta)$ is equal to $\Delta' \bowtie (\mu \cup \text{KEEP}(\tau \cup \Delta))$. This follows directly from the correctness of the migration. $\tau \cup \Delta$ is the set of tuples labeled with the old epoch, while $(\mu \cup \text{KEEP}(\tau \cup \Delta))$ is the same set distributed differently between the machines according to the new mapping. \square

Alg. 4 exploits this equivalence to continue processing tuples throughout migration. Informally, parts (1), (2) and (3) are continuously computed in HANDLETUPLE_1 whereas, (4), (5), (6) and (7) are continuously computed in both HANDLETUPLE_1 and HANDLETUPLE_2 .

Theorem 4.4.6. *Alg. 4 produces the correct and complete output and ensures eventually consistent state for all joiners.*

Proof. First, it is easy to see that the data migration is performed correctly. τ is sent immediately at the very beginning (line 3). Tuples of Δ are sent as they are received (line 20). Finally, the discards are done once the migration is over (line 29). By lemma 4.4.7, the result is the union of:

1. $\tau \bowtie \tau$. This is computed prior to the start of migration as it represents historical state.
2. $(\Delta \bowtie \Delta) \cup (\tau \bowtie \Delta)$. Δ is initially empty. Tuples are only added to it in line 16. Every added tuple gets joined with all previously added tuples to Δ and to all tuples in τ (also in line 16). It follows that this part of the join is computed. τ never changes until the migration is finalized.

Chapter 4. Online Theta Joins

Algorithm 4 Joiner-Epoch Algorithm.

Input: s signal

Initialize: Use HANDLETUPLE_1 to handle incoming tuples.

```
1: procedure MAIN( $s$ )
2:   if First Reshuffler Signal Received then
3:     SEND  $\tau$  for migration.
4:   else if All Reshuffler Signals Received then
5:     Use  $\text{HANDLETUPLE}_2$  to handle incoming tuples.
6:   else if Migration Ended then
7:     Run FINALIZEMIGRATION.
8:   Use  $\text{HANDLETUPLE}_1$  to handle incoming tuples.
```

Input: t an incoming tuple

```
9: procedure HANDLETUPLE1( $t$ )
10:  if  $t \in \mu$  then OUTPUT
11:     $\{t\} \bowtie \Delta'$ ;  $\mu \leftarrow \mu \cup \{t\}$ 
12:  else if  $t \in \Delta'$  then
13:    OUTPUT  $\{t\} \bowtie (\mu \cup \Delta')$ ;  $\Delta' \leftarrow \Delta' \cup \{t\}$ 
14:    OUTPUT  $\{t\} \bowtie \text{KEEP}(\tau \cup \Delta)$ 
15:  else if  $t \in \Delta$  then
16:    OUTPUT  $\{t\} \bowtie (\tau \cup \Delta)$ ;  $\Delta \leftarrow \Delta \cup \{t\}$ 
17:    if  $t \in \text{KEEP}(\Delta)$  then
18:      OUTPUT  $\{t\} \bowtie \Delta'$ 
19:    if  $t \in \text{MIGRATED}(\Delta)$  then
20:      SEND  $\{t\}$  for migration
```

Input: t an incoming tuple

```
21: procedure HANDLETUPLE2( $t$ )
22:  if  $t \in \mu$  then
23:    OUTPUT  $\{t\} \bowtie \Delta'$ ;  $\mu \leftarrow \mu \cup \{t\}$ 
24:  else if  $t \in \Delta'$  then
25:    OUTPUT  $\{t\} \bowtie (\mu \cup \Delta')$ ;  $\Delta' \leftarrow \Delta' \cup \{t\}$ 
26:    OUTPUT  $\{t\} \bowtie \text{KEEP}(\tau \cup \Delta)$ 
```

```
27: procedure FINALIZEMIGRATION
28:  SEND (Ack) signal to coordinator
29:   $\tau \leftarrow \text{KEEP}(\tau \cup \Delta) \cup \mu \cup \Delta'$ 
30:   $\Delta \leftarrow \emptyset$ ;  $\Delta' \leftarrow \emptyset$ ;  $\mu \leftarrow \emptyset$ 
```

3. $\Delta' \bowtie (\mu \cup \text{KEEP}(\tau \cup \Delta))$. Whenever a tuple is added to Δ' (in lines 13 and 25), it gets joined with $\mu \cup \text{KEEP}(\tau \cup \Delta)$ (lines 13, 14, 25 and 26). Whenever a tuple is added to μ (lines 11 and 23), it gets joined with Δ' . Furthermore, tuples added to Δ are joined with Δ' if they are in $\text{KEEP}(\Delta)$

(line 18). τ never changes until the migration ends.

4. $\Delta' \triangleright \Delta'$. Initially Δ' is empty. Tuples get added to it in lines 13 and 25. Whenever a tuple gets added, it gets joined with all previously added tuples (lines 13 and 25).

Therefore, all parts are computed by the algorithm (completeness). Since the analysis covers all the lines that perform a join, it follows that each of the 4 parts of the result is output exactly once (correctness). Thus, the result of the algorithm is correct right after migration is complete. Tuples tagged with the old epoch (τ and Δ) are migrated correctly. Tuples tagged with the new epoch (Δ') are immediately sent to machines according to the new scheme. Therefore, at the end of migration, the state of all joiners is consistent with the new mapping. \square

Theoretical Guarantees Revisited

The guarantees given in Theorem 4.4.1 assume a blocking operator. During migration, it is required that no tuples are received or processed. However, Alg. 4 continuously processes new tuples while adapting. We set the joiners to process migrated tuples at twice the rate of processing new incoming tuples. We show that, under these settings, the proven guarantees hold. It is clear that the amortized cost is unchanged and remains linear because incoming tuples continue to “pay” for future migration costs. The results for competitiveness, on the other hand, need to be verified.

Theorem 4.4.7. *With the non-blocking scheme Alg. 4, the competitive ratio ensured by Theorem 4.4.1 remains 1.25^i .*

Proof. We prove that the numbers of tuples, received during migration, $|\Delta R|$ and $|\Delta S|$, are bounded by $|R|$ and $|S|$, respectively. 1.25 -competitiveness follows immediately (by lemma 4.4.4).

Consider a migration decision after the system has received $|R|$ and $|S|$ tuples from R and S . Let the current mapping be (n, m) . Lemma 4.4.3 asserts that the optimal mapping is one of (n, m) , $(n/2, 2m)$ and $(2n, m/2)$. This is trivially true for the first migration. Since we prove below that $|\Delta R|$ and $|\Delta S|$ are bounded by $|R|$ and $|S|$, this also holds for all subsequent migrations, inductively. Without loss of generality, let the chosen optimal mapping for a subsequent migration be $(n/2, 2m)$. The migration process lasts for $2|R|/n$ time units (by lemma 4.4.5). Alg. 4 processes new tuples at half the rate of processing migrated tuples. Thus, during migration, the operator receives at most $1/2 \cdot (n/2)$ new tuples from R and $1/2 \cdot (2m)$ from S per time unit. Hence, it holds that,

ⁱNotice that Theorem 4.4.7 is based on the assumptions made in Theorem 4.4.1. However, it naturally follows, that if any of the assumptions are relaxed the competitive ratio is changed accordingly as described in section 4.4.2.

$$|\Delta R| \leq \frac{2|R|}{n} \cdot \frac{n}{4} < |R|$$

$$|\Delta S| \leq \frac{2|R|}{n} \cdot m \leq \frac{|S|}{m} \cdot m = |S|$$

where the last inequality holds by lemma 4.4.2 (with the optimal being $(n/2, 2m)$ instead of (n, m)). \square

Towards Fault-Tolerance

Although fault tolerance is orthogonal to the scope of our discussion, this section outlines how to extend the presented dataflow operator to provide fault-tolerance using existing techniques. For topologies with arbitrary operators, FTOpt's [170] fault-tolerance protocol guarantees *exactly-once* semantics (no lost or duplicate tuples). We can easily extend our operator to follow the protocol such that the entire query plan provides end-to-end fault-tolerance. The protocol is established between any two communicating nodes (producer/consumer pairs) in the query plan by splitting the fault-tolerance responsibilities between them. When a consumer takes responsibility of a received tuple, it sends an acknowledgment to the producer. This frees the producer from replaying acknowledged tuples on failures. The consumer can fulfill its responsibility by checkpointing to stable storage. On the other hand, the producer is responsible for replaying unacknowledged tuples on failure. This protocol supports many-to-many producer/consumer relationships.

At a high level, when a node fails, it first recovers its state from the latest checkpoint. Because some tuples may have been processed successfully on a consumer, but their acknowledgment may not have reached the producer before its failure, the recovered node then communicates with the downstream and upstream nodes to identify which tuples to replay. For every communication pair, the consumer provides information about the last seen tuple, and the producer has to replay only the missing portion of the stream. This protocol can provide fault-tolerance during migration as well. The only additional consideration is that communication pairs may vary due to the different migrations, and hence, this information also needs to be preserved.

4.4.4 Equi-Joins Specialization

Monotonic join conditions can enable further optimizations by avoiding to cover empty regions. Fig. 4.1 demonstrates several examples of monotonic join conditions and their corresponding join matrix structure. We focus here on equi-joins because they are common in practice and are very vulnerable to data skew. Our proposed operator can be further specialized for the

case of equi-joins to enable efficient and skew-resilient equi-joins.

Content-Sensitive Partitioning. One common partitioning strategy is key partitioning, where tuples from both streams are partitioned on the key attribute. This ensures that all tuples with the same key are gathered at the same place for join processing. This approach is simple, maximizes locality, and avoids the overhead of replicated messages or storage. Nevertheless, it is highly vulnerable to skew. Hash-key partitioning simplifies the assignment scheme to a one dimensional domain. That is, partitioning the join-matrix \mathcal{M} is reduced to partitioning a set of keys across machines. This coarse grained approach makes load balancing infeasible under skewed distributions. Moreover, it is sensitive to the order of input values, i.e., it is simple to construct an adversarial data input order that would limit parallelism to a single machine at all times. Previous adaptive approaches [155, 39, 117] fall into this class. FLUX adaptively rebalances load by repartitioning keys from one machine to another whenever a machine is overutilized.

Content-Insensitive Partitioning. This chapter has described a *symmetric* partitioning scheme. This symmetric approach is useful for processing arbitrary join predicates or when the join matrix \mathcal{M} is dense. However, in the common class of selective joins, e.g., equi-joins, the symmetric scheme becomes expensive because of covering unnecessary empty regions within the join matrix \mathcal{M} . This induces additional communication and storage replication overhead. In this scenario, a *content-sensitive* partitioning approach would be more suitable as knowledge about key values are utilized to avoid covering unnecessary regions. In the following, we present an *asymmetric scheme* that *only* covers candidate regions and ignores the rest. Moreover it is a hybrid approach that combines the best of both worlds from content-sensitivity and content-insensitivity.

We observe that the join matrix \mathcal{M} under the equi-join predicate has a specific structure. That is, candidate areas are represented as non-overlapping independent rectangular regions. Thus, the optimization problem can be simplified to that of optimizing each region independently.

Independent Partitioning. Fig. 4.1a illustrates an equi-join example depicted within the join matrix model. We observe that (i) each key defines an independent rectangular region which doesn't share any state with other keys, and (ii) there are large portions of empty regions that don't need to be covered. These observations enable enormous savings. We describe an adaptive scheme specialized for equi-joins. The scheme operates as follows: it regularly maintains key frequencies of each stream. Accordingly, each rectangular region defined by its corresponding key is either hash partitioned or divided into congruent rectangular regions using symmetric partitioning. The partitioning decision is made according to the area of the rectangular region. In this case, we are dealing with fully dense regions, since each rectangle represents a fully joinable region. These costs are quadratic in input size and thus when they surpass a specified threshold, it becomes important to maintain load balance by equally

| Query | Join | Predicate |
|-----------|-------------------------------------|-----------|
| E_{Q_5} | $(R \bowtie N \bowtie S) \bowtie L$ | Equi-join |
| E_{Q_7} | $(S \bowtie N) \bowtie L$ | Equi-join |
| B_{NCI} | $L \bowtie L$ | Band-join |
| B_{CI} | $L \bowtie L$ | Band-join |

Table 4.1 – R, N, S, and L correspond to the relations Region, Nation, Supplier, and Lineitem respectively as defined in the TPC-H benchmark.

distributing this workload.

This hybrid approach combines between content sensitivity by coalescing together dense regions defined by each key independently and content insensitivity by dividing each dense region equally across J machines. Thereby, it combines the best of both worlds. Namely, coalescing the workload and avoiding covering vast empty regions. This results in savings in communication and storage costs, and guaranteed distribution of workload across all machines at all times.

Independent Adaptation. Dividing each large key rectangle using symmetric partitioning allows for independent adaptation. Now each rectangle operates and adapts independently among the same set of J machines as previously described.

4.5 Evaluation

Environment. Our experimental platform consists of an Oracle Blade 6000 Chassis with 10 Oracle X6270 M2 blade servers. Each blade has two Intel Xeon X5675 CPUs running at 3GHz, each with 6 cores and 2 hardware threads per core, 72GB of DDR3 RAM, 4 SATA 3 hard disks of 500GB each, and a 1Gbit Ethernet interface. All blades run Solaris 10, which offers Solaris Zones, a native resource management and containment solution. Overall, there are 220 virtual machines available exclusively for our experiments, each with its own CPU hardware thread and dedicated memory resources. There are 20 separate hardware threads for running instances of the host operating system.

Datasets. For the evaluation setup, we use the TPC-H benchmark [6]. We employ the TPC-H generator proposed by [45] to generate databases with different degrees of skew under the *Zipf* distribution. The degree of skew is adjusted 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 build eight databases with sizes 8, 10, 20, 40, 80, 160, 320, and 640GB.

Queries. We experiment on four join queries, namely, two equi-joins, called E_{Q_5} and E_{Q_7} , from

the TPC-H benchmark and two synthetic band-joins. The equi-joins, E_{Q_5} and E_{Q_7} , represent the most expensive join operation in queries Q_5 and Q_7 , respectively, from the benchmark. All intermediate results are materialized before online processing. Moreover, the two band-joins depict two different workload settings. *a)* B_{CI} is a *high*-selectivity join query that represents a computation-intensive workload, and *b)* B_{NCI} is a *low*-selectivity join query that corresponds to 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. Both join queries are described below and all query characteristics are summarized in Table 4.1.

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. To run the testbed, we implement SQUALLⁱⁱ, a distributed online query processing engine built on STORMⁱⁱⁱ, Twitter’s backend engine for data analytics. The engine is based on Java and runs on JRE v1.7. Throughout the discussion, we use four different dataflow operators: (i) STATICMID, a static operator with a fixed (\sqrt{J}, \sqrt{J}) -mapping. This scheme assumes that both input streams have the same size and lies in the center of the (n, m) -mapping spectrum. (ii) DYNAMIC, our adaptive operator, initialized with the (\sqrt{J}, \sqrt{J}) -mapping scheme. (iii) STATICOPT, another static operator with a fixed optimal mapping scheme. This requires knowledge about the input stream sizes before execution, which is practically *unattainable* in an online setting. (iv) SHJ, the parallel symmetric hash-join operator described in [83]. This operator can only be used for equi-join predicates and it is *content-sensitive* as it partitions data on the join key. STATICMID, assumes as a best guess, that the streams are equal in size; hence it has a square grid partitioning scheme, i.e., (\sqrt{J}, \sqrt{J}) . Comparing against STATICOPT shows that our operator does not perform much worse than an omniscient operator with oracle knowledge about stream sizes, which are unknown beforehand. Joiners perform the

ⁱⁱ<https://github.com/epfldata/squall/wiki>

ⁱⁱⁱ<https://github.com/nathanmarz/storm>

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

Note: [*] Overflow to disk.

Table 4.2 – Runtime in secs.

local join in memory, but if it runs out of memory it begins spilling to disk. For this purpose, we integrated the operators with the back-end storage engine BERKELEYDB [143]. We first experimentally verify that, in case of overflow to disk, machines suffer from long delayed join evaluation and performance hits. Then, for a more fair comparison, we introduce more memory resources, such that all operations fit in memory if possible. The heap size of each joiner is set to 2GB. As indexes, joiners use balanced binary trees for band joins and hashmaps for equi-joins. Input data rates are set such that joiners are fully utilized.

4.5.1 Skew Resilience

Table 4.2 shows results for running joins E_{Q_5} and E_{Q_7} with different skew settings of the 10G dataset. It compares the performance of our DYNAMIC operator against the SHJ operator using 16 machines. We report the final execution time. We observe that SHJ performs well under non-skewed settings as it evenly partitions data among machines and does not replicate data. On the other hand, the DYNAMIC operator, distributes workload fairly between machines, but pays for the unnecessary overhead of replicating data. As data gets skewed, SHJ begins to suffer from poor partitioning and unbalanced distribution of data among joiners. Thus, the progress of join execution is dominated by a few overwhelmed workers, while the remaining starve for more data. The busy workers are congested with input data and must overflow to disk, hindering the performance severely. In contrast, the DYNAMIC operator is resilient to data skew and persists to partition data equally among joiners.

4.5.2 Performance Evaluation

We analyze in detail 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 illustrated in Table 4.2, DYNAMIC operates efficiently, whereas STATICMID consistently performs worse. For skewed data, the latter suffers from very high values of ILF, and thus,

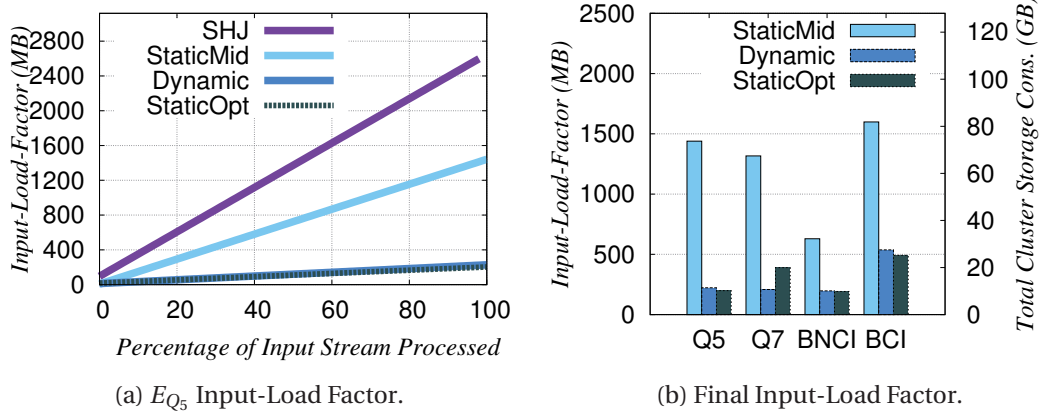


Figure 4.7 – Execution time performance results part I.

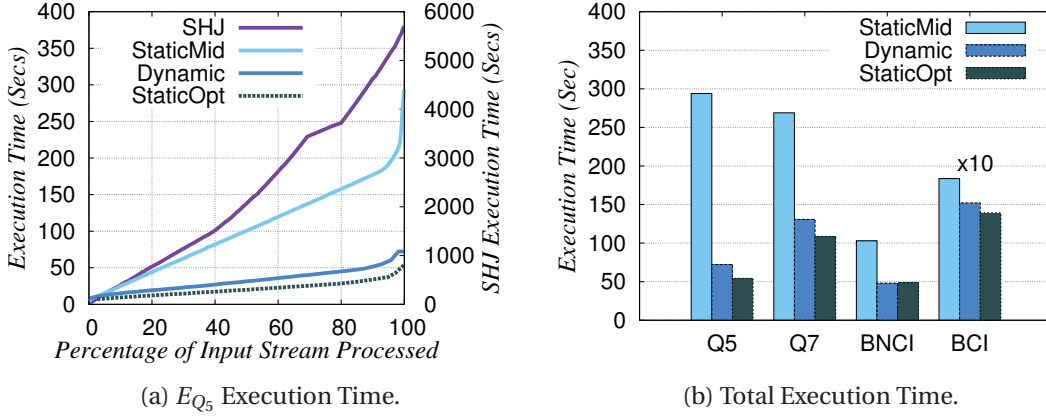


Figure 4.8 – Execution time performance results part II. The 2nd y-axis depicts the results for SHJ only.

overflows to disk, hindering the performance drastically. For a fair comparison, we increase the number of machines to 64 such that STATICMID is given enough resources. Under this setting, STATICMID has a fixed (8,8)-mapping scheme, whereas the optimal mapping scheme for all joins is (1,64). Our results show that DYNAMIC behaves roughly the same as STATICOPT. This is attributed to the fact that DYNAMIC migrates to the optimal mapping scheme at early stages. For completeness, we also include the results for E_{Q_5} and E_{Q_7} using SHJ. The operator overflows to disk due to high data skew.

Input-Load Factor. As described in §4.2.3, different mappings incur different values for the input-load factor. Examining the average input-load factor for each operator shows that the growth rate of the ILF is linear over time. Due to the lack of space, we illustrate this behavior for E_{Q_5} only. Fig. 4.7a plots the maximum size of ILF per machine against the percentage of total

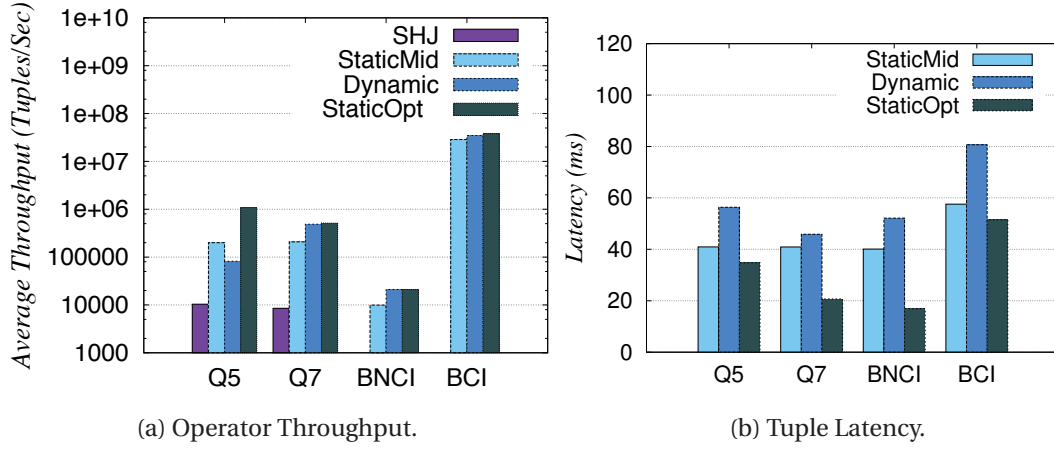


Figure 4.9 – Operator metrics performance results part I.

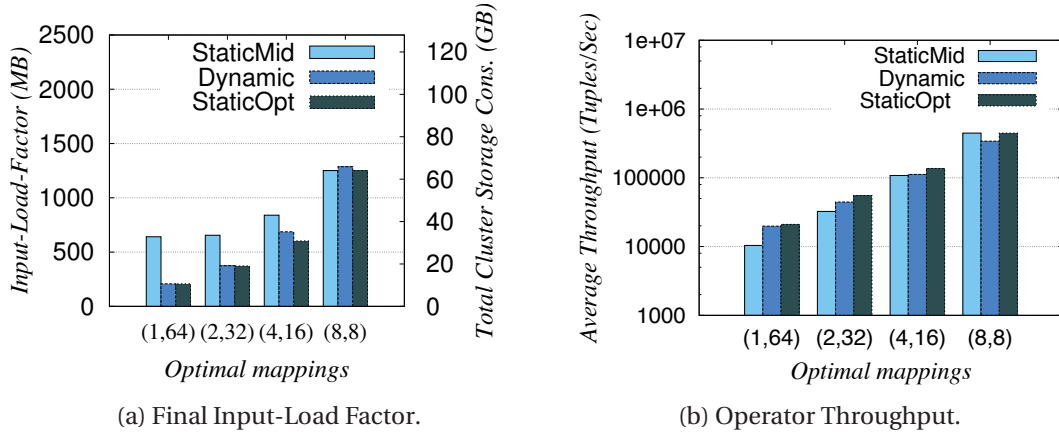


Figure 4.10 – Operator metrics performance results part II.

input stream processed. SHJ and STATICMID suffer from a larger growth rate than DYNAMIC. Specifically, their rates are 27, 14 and 2MB per 1% input stream processed, respectively. The graphs depicted in Fig. 4.7b report on the final average ILF per machine for all the join queries. STATICMID is consistently accompanied with larger ILF values. Its ILF is about 3 to 7 times that of DYNAMIC. The optimal mapping (1, 64) lies at one end of the mapping spectrum and is far from that of STATICMID. And SHJ is up to 13 times that of the other operators.

§4.2.3 also emphasizes the fact that minimizing the ILF maximizes resource utilization and performance. This is due to the fact that higher ILF values also imply (i) unnecessary replicated data stored around the cluster, (ii) more duplicate messages sent congesting the network, and (iii) additional overhead for processing and housekeeping replicated data at each joiner. In what follows, we measure the impact of ILF on overall operator performance.

Resource Utilization. Fig. 4.7b also shows the total cluster storage consumption (GB), as shown on the right axis. STATICMID’s fixed partitioning scheme misuses allocated resources as it unnecessarily consumes more storage and network bandwidth to spread the data. Moreover, it requires four times more machines (64) than DYNAMIC to operate fully in memory (16 machines used in Table 4.2). SHJ could not fully operate in memory even with 64 machines. DYNAMIC performs efficiently in terms of resource utilization. This is essential for cloud infrastructures which typically follow *pay-as-you-go* policies.

Execution Time. Fig. 4.8a shows the execution time to process the input stream for query E_{Q_5} . The other join queries are similar in behavior and we omit them due to the lack of space. Fig. 4.8b shows the total execution time for all the join queries. We observe that execution time is linear in the percentage of input stream processed. The ILF has a decisive effect on processing time. The rigid assignment (8,8) of STATICMID yields high ILF values and leads to consistently worse performance. As ILF grows, the amount of data to process, and hence, processing time increases. However, this performance gap is not large when the join operation is computationally intensive, i.e., B_{CI} in Fig. 4.8b. The execution time for SHJ, shown at the right axis of Fig. 4.8a, is two orders of magnitude more, illustrating that poor resource utilization may push the operator to disk spills, hindering the performance severely. In all cases, the adaptivity of DYNAMIC allows it to perform very close to STATICOPT.

Average Throughput and Latency. Fig. 4.9a shows global operator throughput. For all queries, the throughputs of DYNAMIC and STATICOPT are close. They are at least twice that of STATICMID, and two orders of magnitude more than that of SHJ, except for B_{CI} where the difference is slight. This validates the fact that the ILF has a direct effect on throughput, and that the effect is magnified when overflow occurs. The throughput gap between operators depends on the amount of join computation a machine has to perform (e.g. compare B_{CI} and B_{NCI}). Fig 4.9b shows average tuple latencies. We define latency as the difference between the time an output tuple t is emitted and the time at which the (more recent) corresponding source input tuple arrives to the operator. The figure shows that the operator latency is not greatly affected by its adaptivity. During state migration, an additional network hop increases the tuple latency. DYNAMIC achieves average latency close to that of STATICMID while attaining much better throughput.

Different Optimal Mappings. So far, the join queries we experiment on capture the interesting case of an optimal mapping that is far from the (\sqrt{J}, \sqrt{J}) scheme. As illustrated in Figs. 4.10a, 4.10b, we compare performance under various optimal mapping settings. We achieve this by increasing the size of the smaller input stream. In all cases, DYNAMIC adjusts itself to the optimal mapping at early stages. Fig. 4.10a shows how the input-load factor gap between DYNAMIC and STATICMID decreases as the optimal mapping gets closer to the (\sqrt{J}, \sqrt{J}) -mapping scheme. Similarly, Fig. 4.10b illustrates how the performance gap decreases

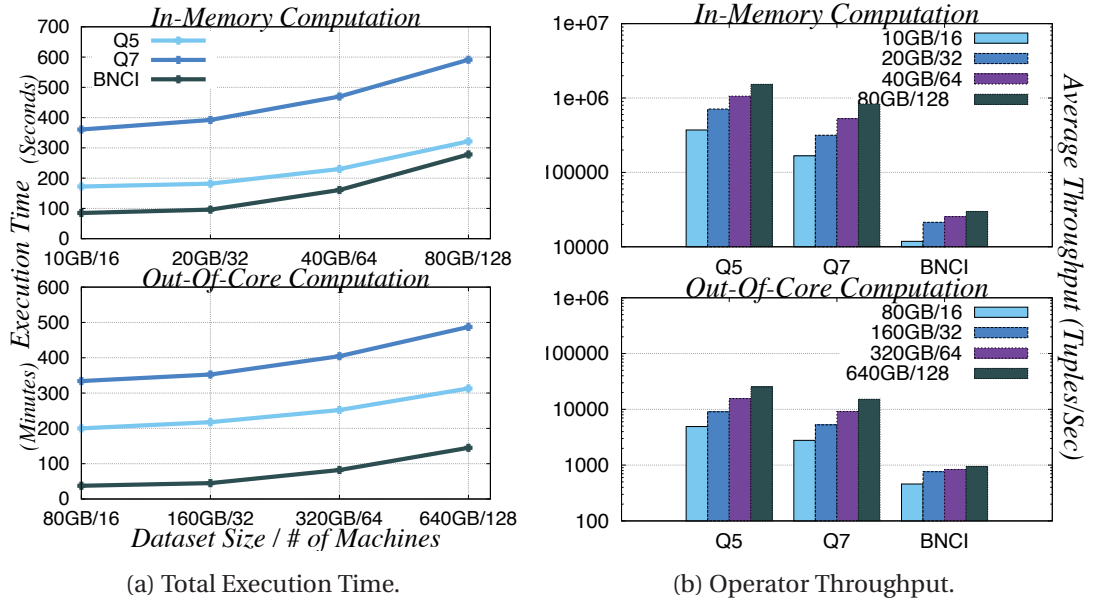


Figure 4.11 – Scalability performance results.

between the two operators. This validates the fact that the input-load factor has a decisive effect on performance. In case of the optimal (\sqrt{J}, \sqrt{J}) -mapping scheme, STATICOPT has the same mapping as STATICMID, whereas DYNAMIC does not deviate from its initial mapping scheme. However, it performs slightly worse because adaptivity comes with a small cost.

4.5.3 Scalability Results

We evaluate the scalability of DYNAMIC. Specifically, we measure operator execution time and throughput as both the data-size and parallelism configurations grow. We evaluate weak scalability on 10GB/16 joiners, 20GB/32 joiners, and so forth as illustrated in the in-memory computation graphs of Figs. 4.11a, 4.11b. Ideally, while increasing the data-size/joiners configuration, the input-load factor and the output size should remain constant per joiner. However, the input-load factor grows, preventing the operator to achieve perfect scalability (same execution time and double average throughput as the data-size/joiners double). For example, for B_{NCI} , under the 20GB/32 configuration, the input stream sizes are 0.68M (million) and 30M tuples, respectively, yielding a $(1, 32)$ optimal mapping scheme with an ILF of $0.68M + 30M/32 \approx 1.61M \cdot size_{tuple}$ 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 with an ILF of $1.36M + 60M/64 \approx 2.29M \cdot size_{tuple}$. In both cases, the output size per joiner is the same (64K tuples). However, the ILF differs by 42% because of the replication of the smaller relation. The ILF for the other two joins does not grow more than 9%. Accordingly, the

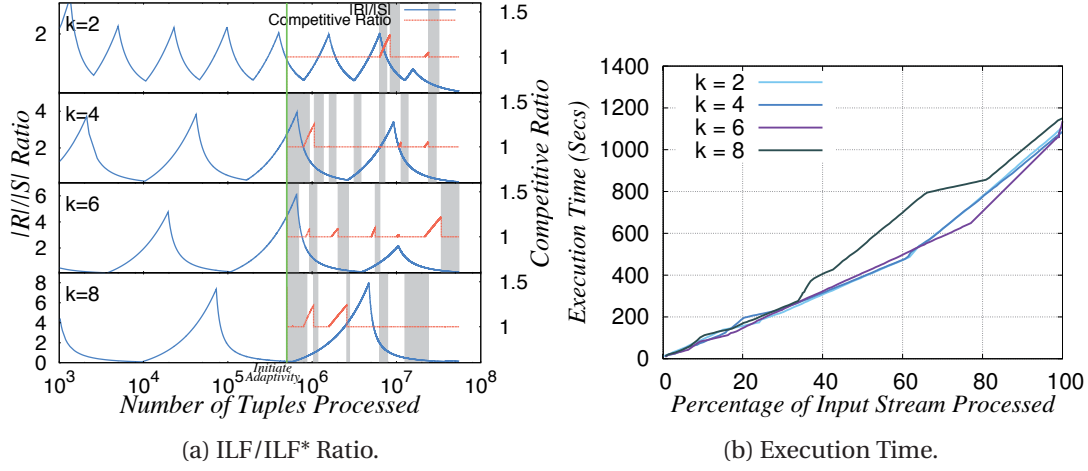


Figure 4.12 – Performance results under fluctuations.

execution time (Fig. 4.11a) and the average throughput (Fig. 4.11b) graphs show that E_{Q_5} and E_{Q_7} achieve almost perfect scalability. In case of B_{NCL} , a joiner processes more input tuples as data grows. Overall, the operator achieves very good scalability taking into account the increase in ILF.

Secondary storage. Out-of-core computation in Figs. 4.11a, 4.11b illustrates performance under weak scalability with secondary storage support. As before, all the queries achieve near optimal scalability, taking into account the increase in ILF. This validates the fact that our system can scale with large volumes of data, and that it works well regardless of the local join algorithm. However, compared to the *in-memory* results (Fig. 4.11a), the performance drops by an order of magnitude. This validates our conclusion that secondary storage is not perfectly suited for high-performance online processing.

4.5.4 Data Dynamics

In order to validate the proven theoretical guarantees, we evaluate the performance of DYNAMIC under severe fluctuations in data arrival rates. We simulate a scenario where the cardinality aspect ratios keep alternating between k and $1/k$ where k is the fluctuation rate. Data from the first relation is streamed into the operator until its cardinality is k times that of the second one. Then, the roles are swapped, by quiescing the first input stream and allowing data to stream in from the second until its cardinality is k times that of the first. This fluctuation continues until the streams are finished. We experiment on an 8G dataset using the *Fluct-Join* query defined below on 64 machines. We run the query under various fluctuation factor, specifically, $k = 2, k = 4, k = 6$ and $k = 8$. We set the operator to begin adapting after it has received at least 500K tuples, corresponding to less than 1% of the total input.

```
SELECT *  
FROM ORDER O, LINEITEM L  
WHERE O.orderkey=L.orderkey  
AND O.shippriority !=5-LOW AND O.shippriority !=1-URGENT
```

Analysis. The first metric of interest is the ILF competitive ratio of DYNAMIC in comparison to an *oracle* that assigns the optimal mapping, and thus optimal ILF*, instantly at all times. Fig. 4.12a plots both the $|R|/|S|$, on the left axis, and the ILF/ILF* ratio, on the right axis, throughout query execution. In the graph, migration durations are depicted by the shaded regions. We observe that the ratio never exceeds 1.25 at all times which validates the result of Theorem 4.4.7. Even under severe fluctuations, the operator is well advised in choosing the right moments to adapt. Fig. 4.12b shows the execution time progress under different fluctuation factor. Although DYNAMIC undergoes many migrations, it persists to progress linearly showing that all migration costs are amortized. This verifies the results of Lemma 4.4.6 and Theorem 4.4.1.

4.5.5 Summary

Experiments show that our adaptive operator outperforms *practical* static schemes in every performance measure without sacrificing low latency. The static schemes emphasize the effect of ILF on resource utilization and performance. This validates the optimization goal of minimizing ILF as a direct performance measure. Our operator ensures efficient resource utilization in storage consumption and network bandwidth that is up to 7 times less than non-adaptive theta-join counterparts. Non-adaptivity causes misuse of allocated resources leading to overflows. Even when provided enough resources, the adaptive operator completes the join up to 4 times faster with an average throughput of up to 4 times more. Adaptivity is achieved at the cost of a slight increase in tuple latency (by as little as 5ms and at most 20ms). Experiments also show that our operator is scalable. Under severe data fluctuations, the operator adapts to data dynamics with the ILF remaining within the proven bounds from the optimum and with amortized linear migration costs. Additionally, the operator, being *content-insensitive*, is resilient to data skew while *content-sensitive* operators suffer from overflows, hindering performance by up to two orders of magnitude.

5 Lago: Online Advanced Analytics

Analytics has seen a paradigm shift from aggregate query processing, e.g., in SQL, to more sophisticated analytics where data practitioners, engineers, and scientists utilize advanced data models to gain insights about the collected data. These analytical tasks include machine learning, statistical analyses, scientific computation, and graph computations. Currently, a wide range of tools and environments for expressing and optimizing such workloads have evolved. These include systems specialized for machine learning tasks such as MLlib [131] and SystemML [78]; platforms dedicated for graph processing such as GraphChi [113], GraphLab [119], Pregel [123] and PowerGraph [80]; low-level autotuned kernels such as Spiral [149] for linear transformations; and Riot [192] for out-of-core statistical analysis. However, the existing tools lack support for dynamic datasets. Most datasets evolve through changes that are small relative to the overall dataset size. For example, the activity of a single customer, like his purchase history or review ratings, represents only a tiny portion of the overall collected data corpus. Recomputing data analytics on every (moderate) dataset change is far from efficient. An alternative approach, called Incremental View Maintenance, combines pre-computations with incoming Δ changes to provide a computationally cheap method for updating the final result. IVM [30, 107, 87] of relational calculus is well known in the Databases literature. Unfortunately, these techniques are not compatible with other domains. Many of the advanced analyses boil down to linear-algebra expressions over matrices [78, 105, 38, 173]. Matrix algebra represents a concrete substrate for analytical tasks, machine learning, scientific computation, and graph algorithms. Many machine learning algorithms, including regression, recommendations, and matrix factorizations, are representable as matrix manipulation operations [78]. Moreover, recent research [105, 38, 173] suggests that modelling graph analytics using matrix operations results in better parallelization efficiency, e.g., coarse grained parallelism, and higher productivity, e.g., using a simpler level of abstraction.

5.1 Challenges and Contributions

This chapter presents techniques and tools that support Incremental View Maintenance of advanced analytics represented as matrix algebra. We now present the structure of this chapter while outlining our main contributions:

1. **Lago: Automatic IVM Derivation:** Lago (Section 5.3) is a unified modular compiler framework that supports the IVM of a broad class of linear algebra programs. Lago automatically derives and optimizes incremental trigger programs of analytical computations, while freeing the user from erroneous manual derivations, low-level implementation details, and performance tuning. Lago extends and builds upon previous work [140] that presents a novel technique that captures Δ changes as low-rank matrices (Section 5.2.1). Low-rank matrices are representable in a compressed factored form that enables converting programs that utilize expensive $\mathcal{O}(n^3)$ matrix operationsⁱ such as matrix-matrix multiplication and matrix-inverse, to trigger programs that evaluate delta expressions with asymptotically cheaper $\mathcal{O}(n^2)$ matrix operations, e.g., matrix-vector multiplication. Lago utilizes the low-rank property and automatically propagates it across program statements to derive an efficient trigger program. Moreover, Lago extends its support to other applications and domains of different semi-ring configurations, e.g., graph applications.
2. **Framework and Components Synergy:** The Lago framework is based upon several synergistic components: 1) First, Lago provides a domain-specific language (Section 5.3.2) that supports basic linear algebra operations including multiplication, addition, inverse, transpose, etc. This establishes a common ground that decouples the high-level representation of the program from IVM derivation, optimization, and domain representation, e.g., semi-ring configuration. The language is designed to be succinct to maintain simplicity while preserving sufficient expressiveness, i.e., other constructs can be defined as compositions of the core language. 2) Secondly, we present a set of domain-specific transformation rules that allows for delta derivation, simplification, and cost-based optimization of matrix algebra programs (Section 5.3.3). 3) Thirdly, we leverage and infer matrix-expression properties and meta-information including data type, dimensions, cost, symmetry, etc, to enable IVM and achieve high performance. Examples in this chapter demonstrate how dimensions are used to guide cost-based optimization; how symmetry enables further transformations; and how data and semi-ring types permit specialization opportunities during code generation (Section 5.3.4). Such information enables orders of magnitude

ⁱMore precisely, the complexity of matrix multiplication is $\mathcal{O}(n^\gamma)$ where $2 \leq \gamma \leq 3$. For all practical reasons, the complexity of matrix multiplication implementations, e.g., using BLAS [179] has cubic cost $\mathcal{O}(n^3)$. Other algorithms, such as Coppersmith-Winograd and its successors, suggest better exponents of $2.37 + \epsilon$; however, these algorithms are only applicable for astronomically large matrices. Our incremental techniques remain relevant as long as matrix multiplication stays asymptotically worse than quadratic time. Note that the asymptotic lower bound is $\Omega(n^2)$ operations because it needs to process at least all $2n^2$ entries.

performance improvements as demonstrated in Section 5.6. 4) Finally, in Section 5.3.5, we show how all these components are wired up all together to construct Lago. In particular, we present a four phase compilation strategy that automatically transforms an input matrix program to an efficient trigger program that is amenable to dynamic datasets.

3. **Use cases & Evaluation:** In sections 5.4 and § 5.6, we present and evaluate the IVM of several practical use case examples including computing linear regression models, gradient descent, and all-pairs graph reachability or shortest path computations. The evaluation results demonstrate orders of magnitude better performance in favor of derived trigger programs in comparison to simple re-evaluation.

5.2 Incremental Computation Δ

In the following discussion we use the following terminology: (i) column vectors are denoted by lower case letters \mathbf{x} , matrices are represented as upper case letters \mathbf{X} , and subscripts are used to name the respective vectors and matrices, (ii) $\Delta_{\mathbf{X}}(\mathbf{Y})$ represents the delta function of expression \mathbf{Y} given changes to matrix \mathbf{X} and $\delta_{\mathbf{Y}}$ represents the delta variable that evaluates the delta expression $\Delta_{\mathbf{X}}(\mathbf{Y})$. (iii) and finally, unless otherwise stated, arithmetic matrix operations are denoted by $+$ and \cdot for addition and multiplication respectively.

Most datasets evolve through changes that are small relative to the overall dataset size. For example, a social network graph evolves through connections that are relatively small in comparison to the entire graph size. Recomputing data analytics on every slight dataset change is far from efficient. Incremental View Maintenance [30, 107, 87] (IVM) studies the incremental maintenance of relational queries. IVM trades off storage in favor of cheaper computations. The main idea is to confine the re-evaluation to the changes affected by the incremental updates only. Then, they are used to update materialized views of the precomputed results. Within the Databases literature, several approaches [30, 107, 87] have been proposed to achieve this. Most notably, DBToaster [107] achieves orders of magnitude better performance on SQL queries in comparison to traditional re-evaluation. However, these approaches are not applicable to matrix programs. To demonstrate this, consider the simple example of computing matrix powers. Matrix powers play an important role in many different domains including evaluating the stochastic matrix of a Markov chain after k steps, solving systems of linear differential equations using matrix exponentials, answering graph reachability queries after k hops. Fig. 5.1a demonstrates an example of computing the 8th power of the input matrix A . The program requires computing 3 costly $\mathcal{O}(n^3)$ matrix-matrix multiplications to evaluate the result. Now, consider a trigger program that updates the final result given a single entry change ΔA to the input matrix A . For explanatory reasons, Fig. 5.1b gives a simplistic representation of such a trigger program where it computes the delta expression for each of the intermediate variables B , C , and D , respectively. Then finally, these materialized views are updated with

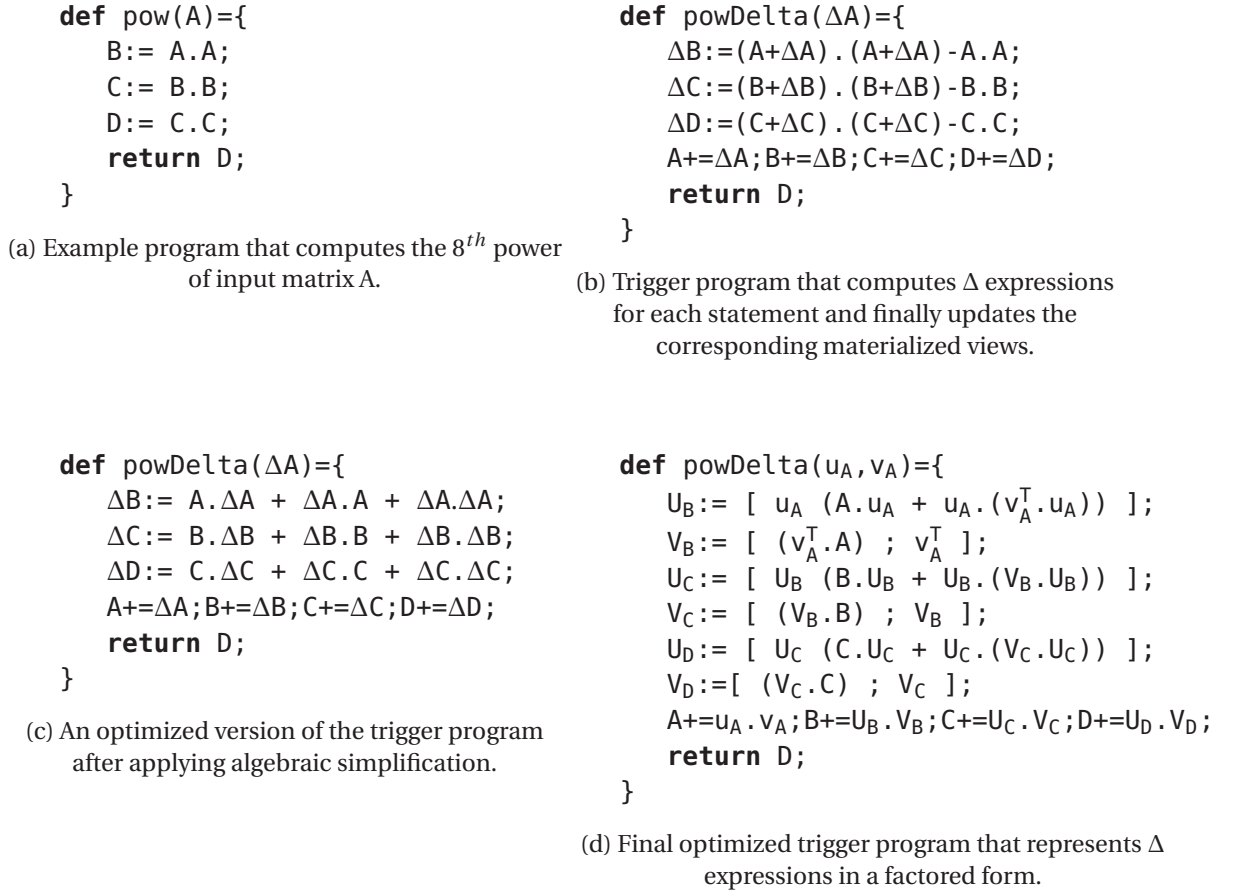


Figure 5.1 – Deriving the trigger program for the matrix powers program A^8

the corresponding delta expressions, e.g., $B+=\Delta B$. Furthermore, when the expressions are expanded algebraically utilizing the associative and distributive laws of matrix addition over multiplication, one could deduce the more simplified expressions illustrated in Fig. 5.1c.

On relatively small changes, one could imagine that by confining the computation to the deltas, we could achieve better performance in comparison to re-evaluation. Unfortunately, this is not the case. As depicted in Fig. 5.2, consider a single entry change ΔA in A. As the figure illustrates, dark cells correspond to entry changes where as white cells correspond to the neutral value, i.e., no change. We can easily compute ΔB in $\mathcal{O}(n^2)$ time, as there is only one single entry in ΔA . After the multiplication, the resulting ΔB matrix has entry changes on a single row and a single column. Similarly, computing ΔC can be done in $\mathcal{O}(n^2)$ time, as one only needs to multiply the two vectors from ΔB with full matrices. However, this is not the case anymore when computing ΔD . ΔC has changes all over its matrix entries. When it is used in the

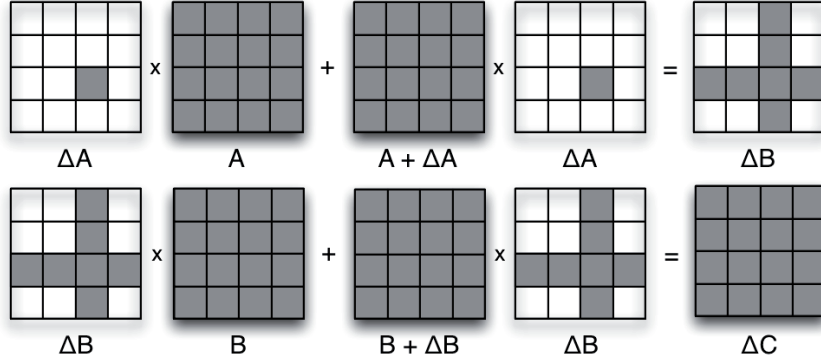


Figure 5.2 – A single data-entry change ΔA in the input A can result in whole matrix perturbations of subsequent Δ expressions. ΔB has changes in a row and a column, whereas ΔC has changes all over all the entries.

subsequent statement to compute ΔD , full fledged $\mathcal{O}(n^3)$ matrix multiplications are required. This renders incremental computation useless in comparison to naive re-evaluation. The above example shows that linear algebra programs are, in general, sensitive to input changes. Even a single entry change in the input can cause an avalanche effect of perturbations, quickly escalating to full matrix perturbations, even after executing only two statements.

5.2.1 The Delta Δ Representation

Until now, we have stored the results of Δ expressions in full matrices. However, one can realize that this representation is highly redundant and that Δ s are usually characterized by having low ranks. Capturing this information is important, as it enables representing the Δ expressions in a packed factored form which compacts storage and greatly reduces the computation cost of its evaluation. The matrix rank- k is defined as the maximum number of linearly independent rows or columns in the matrix.

Consider a matrix A being updated with ΔA , i.e., $A + \Delta A$. If ΔA is a single entry change then it is a rank-1 matrix. Moreover, the expression $A + \Delta A$ represents a rank-1 update. In fact, a rank-1 update can represent updates of a single row/column or even several rows/columns that are linearly dependent to each other. A rank-1 matrix can be represented in a compressed compact form as an outer product of two vectors $\Delta := uv^T$ rather than a full matrix. To demonstrate this, suppose that matrix A has dimensions 3×3 and the single entry change ΔA adds the value

c at index $[2, 2]$ of matrix A . This change can be represented in the factored form as follows:

$$\Delta A := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & c \end{bmatrix} := u \cdot v^T := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & c \end{bmatrix}.$$

Similarly a row change or a column change at $[2, _]$ or $[_, 2]$ can be represented as follows respectively:

$$\Delta A := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ c_0 & c_1 & c_2 \end{bmatrix} := u \cdot v^T := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} c_0 & c_1 & c_2 \end{bmatrix},$$

$$\Delta A := \begin{bmatrix} 0 & 0 & c_0 \\ 0 & 0 & c_1 \\ 0 & 0 & c_2 \end{bmatrix} := u \cdot v^T := \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}.$$

In general, rank- k matrices can represent more general update patterns as they can be represented as a sum of k rank-1 matrices.

Let us illustrate the benefits of this factored form in the previous example. Consider a rank-1 update $\Delta A = u_A v_A^T$, where u_A and v_A are column vectors. One can compute $\Delta B := u_A (v_A^T A) + (A u_A) v_A^T + (u_A (v_A^T u_A)) v_A^T$ as a sum of three outer products. The evaluation order enforced by these parentheses results in matrix-vector and vector-vector multiplications only. Thus, the evaluation of ΔB requires $\mathcal{O}(n^2)$ operations only. Moreover, rather than representing the delta expressions as a sum of outer products, we represent them in a more compact vectorized form for performance, storage, and presentation reasons. Generally, a sum of k outer products is equivalent to a single product of two matrices with dimensions $(n \times k)$ and $(k \times n)$, which are obtained by horizontally/vertically stacking the corresponding vectors together as follows:

$$u_a \cdot v_a^T + u_b \cdot v_b^T + u_c \cdot v_c^T := \begin{bmatrix} u_a & u_b & u_c \end{bmatrix} \begin{bmatrix} v_a^T \\ v_b^T \\ v_c^T \end{bmatrix} := U V$$

where U and V are block matrices with dimensions $(n \times 3)$ and $(3 \times n)$ respectively. Following the same structure, we can represent ΔB in the factored form $U_B V_B$ (with rank-2) as derived in Fig. 5.1d:

$$\begin{aligned} \Delta B &:= u_A \cdot (v_A^T \cdot A) + (A \cdot u_A + u_A \cdot (v_A^T \cdot u_A)) \cdot v_A^T \Rightarrow \\ U_B &:= \begin{bmatrix} u_A & (A \cdot u_A + u_A \cdot (v_A^T \cdot u_A)) \end{bmatrix} \\ V_B &:= \begin{bmatrix} (v_A^T \cdot A) & v_A^T \end{bmatrix} \end{aligned}$$

This factored representation is forward substituted further down the program to derive the Δ

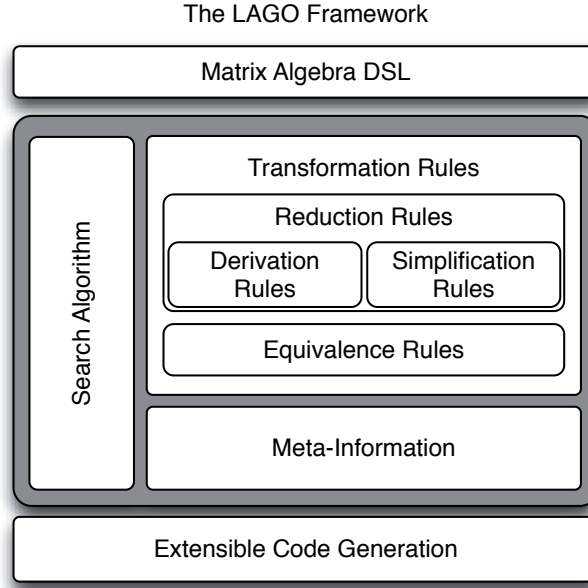


Figure 5.3 – The architecture of the Lago framework.

expressions for each of C and D as depicted in Fig. 5.1d.

In summary and without loss of generality, we capitalize on the low-rank structure of delta matrices by representing a delta expression $\Delta_{n \times n}$ of rank k as a product of two matrices with dimensions $(n \times k)$ and $(k \times n)$, where $k \ll n$. This allows for efficient evaluation of subsequent delta expressions without performing expensive $\mathcal{O}(n^3)$ operations; instead, only $\mathcal{O}(kn^2)$ operations are computed. The benefit of incremental processing diminishes as k approaches n .

5.3 The LAGO Framework

In the previous section, we introduced the concept of incremental computation for matrix algebra and the ability to derive efficient trigger programs by representing updates in a factored form through exploitation of their low rank structure. In this section, we discuss how to automatically derive those trigger programs. One could assume a manual approach in dealing with this problem, however the developer has to put effort into deriving the incremental program, then optimizing it to ensure low cost computation, then finally writing down the code for the trigger program. This is a long and tedious process that includes *a)* delta derivation which is error prone, *b)* optimization which requires simplification, cost-based rewrites, and delicate ordering of operations, and *c)* writing the final trigger program code which requires

careful consideration, e.g., evaluating the delta expressions using the precomputed results before updating the views. We propose offloading all of these responsibilities to Lago, a compiler framework dedicated for deriving, optimizing, and generating trigger code for various underlying processing substrates, thereby freeing the user from erroneous manual derivations, optimization, and low-level implementation details.

5.3.1 Architecture

In this section, we present the architecture of the Lago framework, then we describe its underlying components in detail, and finally, we discuss how all these components are wired together to achieve our goals. The main tasks of the Lago framework are as follows: 1. accepting an input matrix program; 2. deriving the incremental Δ expressions for the statements; performing simplification and cost-based optimizations to optimize the derived expressions; and finally 3. generating the output trigger program code for the underlying system using the derived Δ expressions. Fig. 5.3 gives an overview of the Lago architecture.

1. First, section 5.3.2 presents the domain-specific language used to describe matrix programs in the framework. It includes a restricted set of domain-specific operations specialized for matrix algebra that is independent of the application domain.
2. To derive the incremental program, Lago needs a set of reduction rules that symbolically derive the incremental expressions from the input program and those that simplify the derived expressions. Afterwards, equivalence rules are applied whenever optimization is required. These are called transformation rules (Section 5.3.3). A search module is required to navigate the search space of functionally equivalent programs created by the optimization rules. Different search algorithms can be utilized for different workloads. Similar to DBMSs, various flavours of search algorithms can be employed, such as brute force, DP-programming, or randomized algorithms [98]. However, the discussion about search strategies is orthogonal to this chapter. To evaluate the cost of candidate programs, one needs to estimate their running costs. Similar to how Database Management Systems (DBMSs) estimate query-plan costs using cardinality and selectivity information, Lago leverages meta-information (Section 5.3.4) for matrix programs. The meta-information encapsulates various properties associated to matrix expressions, such as dimensions, structure, symmetry, rank, etc. Moreover, Lago defines inference rules to derive this meta-information for candidate programs whenever possible. This information helps in estimating program costs and in leveraging specialized back-end implementations.
3. Finally, Lago generates trigger code for the underlying processing substrate using code generation. Code generation is extensible, in the sense that one can easily use various code generators for different environments, e.g., Octave, R, Spark, SystemML, etc. The main task

| | | | |
|-----|-------|--|---------------------------------|
| m | $::=$ | $m \cdot m$ | – Matrix-Matrix Multiplication |
| | | $m + m$ | – Matrix Addition |
| | | m^T | – Matrix Transpose |
| | | m^{-1} | – Matrix Inverse |
| | | $m \Rightarrow m$ | – Matrix Concatenation |
| | | vect s | – Row Matrix Construction |
| | | diag $[s][s]$ | – Diagonal Matrix Construction |
| | | let $x = m_1$ in m_2 | – Let binding |
| | | iterate $[s](m)(x \Rightarrow m)$ | – Matrix Iteration |
| | | x | – Matrix Identifier |
| s | $::=$ | $s \text{ bop } s$ | – Scalar Binary Operation |
| | | cols (m) | – Number of Columns of a Matrix |

Figure 5.4 – The core Lago DSL divided into two main classes, i.e., matrix and scalar operations.

in this phase is to generate code for the materialized precomputed results and updating them with their corresponding optimized Δ expressions.

Next, we discuss each of these components in detail, then we describe how they interact with each other to assemble the Lago framework.

5.3.2 Lago DSL

Many sophisticated data analytics programs, including machine learning, graph algorithms, and statistical programs, express computation using matrices and vectors using a high-level abstraction. Lago exposes a domain-specific language (DSL) that expresses such program formulations. The DSL is formulated using standard matrix manipulation primitives excluding elementwise operationsⁱⁱ.

Lago adopts a *functional* approach in the language design. This choice is motivated by the design of languages like relational algebra and Monad algebra [35, 36] in the DB community. Functional programming is a popular paradigm that treats computation as the evaluation of mathematical functions while avoiding state mutation. Since the domain in hand is also mathematical, this paradigm fits well and the inherited benefits are manifold. Most problems that commonly arise in imperative languages from mutable state and side effects are elimi-

ⁱⁱElementwise operations could not propagate factorized expressions down the program. For instance, the expression $X \cdot u \cdot v^T$, could not exploit the low rank structure and be further factorized into UV . That said, the Lago core language can be extended with additional operations only if they can satisfy this requirement, i.e., when updated with a low rank expression the resulting delta expression maintains a low rank that can be represented in a compact factorized form, e.g. the Woodbury formula for matrix inverses.

| Matlab | R | Lago |
|---------------|-------------------|-----------------------|
| $A * B$ | $A \% \% B$ | $A \cdot B$ |
| $A + B$ | $A + B$ | $A + B$ |
| A' | $t(A)$ | A^\top |
| $[A, B]$ | $cbind(A, B)$ | $A \Rightarrow B$ |
| $[A; B]$ | $rbind(A, B)$ | $A \Downarrow B$ |
| $ones(n, m)$ | $matrix(1, n, m)$ | ones $[n, m]$ |
| $zeros(n, m)$ | $matrix(0, n, m)$ | zeros $[n, m]$ |
| $eye(n)$ | $diag(n)$ | id $[n]$ |

Table 5.1 – Equivalent operations in Matlab, R, and Lago respectively. Fig. 5.5 defines the extended operations.

nated. This plays a critical role in performing optimizations. In particular, transformations and their compositions preserve semantics. This eliminates worries about overall program correctness. Moreover, given the mathematical nature of the DSL, transformation rules and meta-information inference rules are much easier defined, as later discussed in Section 5.3.2 and Section 5.3.4. Alternatively, other declarative languages proposed in the literature, such as SystemML [78], take an imperative approach that supports generic control flow and mutable state. Mutable state enables better runtime performance yet makes reasoning and optimization much harder.

Core Language

Another important design choice that drives the language design is to keep the core language succinct enough while maintaining expressiveness that supports a wider range of linear algebra operations through composition. This keeps the language simple, which in turn keeps all the transformation and inference rules at a simple maintainable level. Fig. 5.4 presents Lago's core language grammar which includes matrix multiplication, matrix addition, transpose, matrix inverse, horizontal concatenation (stacking), matrix construction, diagonal matrix construction, let binding, and declaring matrix identifiers respectively. **vect** $[s_1](s_2)$ constructs a $1 \times s_1$ matrix with the constant value s_2 . **diag** $[s_1][s_2]$ creates an $s_1 \times s_1$ matrix with diagonal elements of value s_2 . The rest of operations are self-explanatory. Additionally, we define binary operations on scalars and computing the columns of a matrix.

These constructs are sufficient for expressing non-iterative matrix operations. However, supporting iterative computation is a challenging undertaking. For example, the declarative language presented in SystemML [78] uses imperative constructs such as while loops. Alternatively, to support iterative computation without using imperative loops or mutable states we present the **iterate** $[s](m)(x \Rightarrow m)$ construct which allows us to perform step-by-step computations. The construct is defined by specifying the number of iterations s , the matrix

| | |
|---|---|
| fill [r, c](s) | $m_1 \Downarrow m_2$ |
| $:= \mathbf{vect}[r](1)^\top \cdot \mathbf{vect}[c](s)$ | $:= (m_1^\top \Rightarrow m_2^\top)^\top$ |
| rows (m) | id [n] |
| $:= \mathbf{cols}(m^\top)$ | $:= \mathbf{diag}[n][1]$ |
| $k \cdot m$ | ones [r, c] |
| $:= \mathbf{diag}[\mathbf{rows}(m)][k] \cdot m$ | $:= \mathbf{fill}[r, c](1)$ |
| $m_1 - m_2$ | zeros [r, c] |
| $:= m_1 + -1 \cdot m_2$ | $:= \mathbf{fill}[r, c](0)$ |

Figure 5.5 – Syntactic sugar: Examples of additional operations defined using compositions of the Lago DSL.

m value for the initial step, i.e., neutral value, as well as a function, $x \Rightarrow m$, which computes the current step given the accumulator computed from the previous steps. Notice how the **iterate** operator relates to the functional fold operator. It is important to note that the **iterate** construct represents syntactic sugar for recursive functions. An initial program with the **iterate** operator is first expanded into simpler core language constructs using the simplification rules described next.

Extensions

Various matrix manipulation operations can be defined as syntactic sugar over the core language. This means that there is no need to extend the core language with further redundant operations, which in turn complicates the language, the transformations, the reasoning power, the search space and eventually the modularity of the framework. Instead, one can define these operations in terms of compositions of the core language constructs. Fig. 5.5 demonstrates the expressiveness of the core Lago DSL and Table 5.1 illustrates some examples of equivalent operations in Matlab, R, and Lago using compositions of the small set of core language operations.

Semiring Configurations. Different domains and applications can be built on top of matrix algebra using various semiring configurations. One domain example that makes use of matrix algebra and semirings is graph computation. To explain this further, let's define a semiring first:

Definition 5.3.1. *Given a set S and two binary operations \oplus, \otimes called addition and multiplication respectively, a semiring $\langle S, \oplus, \otimes \rangle$ is an algebraic structure, such that $\langle S, \oplus \rangle$ is a commutative monoid with the identity element 0 , $\langle S, \otimes \rangle$ is a monoid with the identity element 1 , left and right multiplication \otimes distributes over addition \oplus , and multiplication by 0 yields back 0 .*

$$\begin{array}{lcl}
 m & ::= & m \otimes_{\mathcal{R}} m \\
 & | & m \oplus_{\mathcal{R}} m \\
 m_1 \cdot m_2 & ::= & m \otimes_{\mathcal{N}} m \\
 m_1 + m_2 & ::= & m \oplus_{\mathcal{N}} m
 \end{array}$$

Figure 5.6 – Matrix addition and multiplication in the core Lago DSL generalized for semirings

iterate[k](**id**)(acc=> G · acc + **id**)

Figure 5.7 – Program \mathcal{P} represents all-pairs Graph Reachability or Shortest Path after k-hops depending on the semiring configuration.

Semirings & Graphs. Graphs are among the most important abstract data structures in computer science. The algorithms that operate on them are critical to a wide variety of applications including bioinformatics, computer networks, and social media. The vast growth in graph data has forced a shift to parallel computing for executing graph algorithms. Implementing parallel graph algorithms while achieving good parallel performance is a difficult task as it requires fine grained synchronization [38]. Recently, there has been an interest in addressing this challenge by exploiting the duality between the canonical representation of graphs as abstract collections of vertices and edges and a sparse adjacency matrix representation [105, 38, 173]. Furthermore, there is a duality between the fundamental operations on graphs, Breadth First Search (BFS), and the fundamental operation of matrices, matrix multiplication. The benefits of representing graph algorithms as matrices are manifold [105, 38, 173]. Firstly, this linear algebraic approach is widely accessible to scientists and engineers who may not be formally trained in computer science. Secondly, higher performance can be achieved, as parallelizing graph algorithms can now leverage decades worth of research on parallelizing matrix operations, coarse grained parallelism, and optimization with regards to the memory hierarchy. And finally, it leverages productivity and ease of implementation. The common primitive operations used are the numerical addition and multiplication which define a semi-ring $\langle S, \oplus, \otimes \rangle$ where $S \in \{\mathbb{R}\}$, $\oplus = +$, $\otimes = \times$. Many graph problems can be articulated as matrix algebra programs under different semi-ring semantics. For instance, computing all-pairs graph reachability or shortest path after k hops can be expressed as program \mathcal{P} depicted in Fig. 5.7. The semiring configuration defines the semantics of the program. For example, program \mathcal{P} with the Boolean semiring $\langle \{0, 1\}, \vee, \wedge \rangle$ configuration expresses the k-hop reachability program. Similarly, with the tropical semiring $\langle \mathbb{R}, \min, + \rangle$ [38] configuration, program \mathcal{P} expresses the k-hop shortest path program.

IVM & Semirings. Lago exposes a DSL that supports high level matrix operations. This directly allows us to represent graph programs. Moreover, all the primitives that we have previously

| | |
|--|---------------|
| $\Delta_x(m_1 + m_2) \rightarrow \Delta_x(m_1) + \Delta_x(m_2)$ | DELTA-ADD |
| $\Delta_x(m_1 \cdot m_2) \rightarrow$ $\Delta_x(m_1) \cdot m_2 + m_1 \cdot \Delta_x(m_2) + \Delta_x(m_1) \cdot \Delta_x(m_2)$ | DELTA-MULT |
| $\Delta_x(m^\top) \rightarrow (\Delta_x(m))^\top$ | DELTA-TRANS |
| $\Delta_x(m^{-1}) \rightarrow (m + \Delta_x(m))^{-1} - m^{-1}$ | DELTA-INV |
| $\Delta_x(y) \rightarrow \delta_y \quad \text{if } x = y$ | DELTA-VAR-EQ |
| $\Delta_x(y) \rightarrow \mathbf{zeros}[\mathbf{rows}(y), \mathbf{cols}(y)] \quad \text{if } x \neq y$ | DELTA-VAR-NEQ |
| $\Delta_x(m_1 \rightrightarrows m_2) \rightarrow \Delta_x(m_1) \rightrightarrows \Delta_x(m_2)$ | DELTA-STACK |
| $\Delta_x(\mathbf{let } x = m_1 \mathbf{ in } m_2) \rightarrow$ $\mathbf{let } \delta_x(y) = \Delta_y(m_1) \mathbf{ in } \Delta_x(m_2)$ | DELTA-LET |
| $\Delta_x(\mathbf{vect}[c](s)) \rightarrow \mathbf{zeros}[1, c]$ | DELTA-VECT |
| $\Delta_x(\mathbf{diag}[c][s]) \rightarrow \mathbf{zeros}[c, c]$ | DELTA-DIAG |

Figure 5.8 – Delta Δ derivation rules for the core language constructs. The **iterate** construct is first unfolded using the simplification rules in the appendix before applying Δ rules on it. Moreover, the Δ rule for matrix inverse enables the cheaper Woodbury formula as explained in the subsequent examples in section 5.4.1.

presented to support the incremental view maintenance of matrix expressions naturally follow under the different semiring definitions. For example, as graphs evolve with time, one can model new connections in the graph as ΔG expressions added to the original adjacency matrix G^{iii} .

Deriving the Δ expressions and trigger programs is only concerned with the abstract representation of matrices and their transformations, and is independent from the semiring definition. However, this information is useful later on during the code generation phase for specialization, as demonstrated in the section 5.6. The semiring information can be expressed using the core language except for matrix inverse. As illustrated in Fig. 5.6, we generalize the Lago core language by replacing matrix addition and multiplication with two meta-operators \oplus_R and \otimes_R parameterized by a semi-ring \mathcal{R} instance. For example, \oplus_N and \otimes_N are concrete instances of the meta-operators with the arithmetic semiring \mathcal{N} parameter.

5.3.3 Transformation Rules

Definition 5.3.2. *The $\Delta_x(m)$ operator symbolically derives the delta Δ of expression m with respect to variable (or matrix) x . Derivation of the delta expressions and their optimizations are achieved by recursively applying delta transformation rules on the expression m until all Δ operators are omitted.*

ⁱⁱⁱNote that Δ changes represented as additions are naturally defined within the semiring, however deletions depend on the availability of an additive inverse, for example under the boolean semiring, the additive inverse does not exist and thus we cannot model deletions.

There are two types of transformation rules: First, reduction rules which are used to derive the Δ operators and to perform simplifications on the derived expressions. Second, to further perform cost-based optimizations, Lago relies on a set of equivalence rules that create a space of functionally equivalent programs which are passed to the search algorithm in order to find a program with optimal cost. Transformation rules are responsible for constructing the search space of programs. It is very important that transformation rules preserve program semantics.

For illustration purposes, consider a 2-hop instance of the Graph program in Fig. 5.7 yielding the following expression^{iv}:

$$G.G + G.id + id$$

We will continue using this simple running example throughout the following subsections.

Reduction Rules

These are rules in the form of $lhs \rightarrow rhs$, where it always reduces a matched expression from the left-hand-side to the right-hand-side. There are two classes of reduction rules, in particular, derivation and simplification rules which are explained next.

Derivation Rules. This class of reduction rules are used to derive the delta expressions, Δ operators are expanded and evaluated recursively. Using the distributive and associative properties of common matrix operations, we demonstrate the set of delta derivation rules for the core language as depicted in Fig. 5.8. The rules are applied recursively until all deltas of expressions are evaluated, i.e., no more matching derivation rules exist. To illustrate this, given our running example, consider that graph G is evolving with Δ_G changes and that we would like to evaluate the following expression:

$$\Delta_G(G.G + G.id + id)$$

We notice, that the Δ operator is applied over an entire expression that can be reduced by the derivation rules. First, after applying the DELTA-ADD rule, the expression becomes:

$$\Delta_G(G.G) + \Delta_G(G.id) + \Delta_G(id)$$

Furthermore, applying the DELTA-MULT rule yields:

^{iv}Notice that we omit the **id** in $G.G.id$. This is only meant to simplify the following flow and avoid redundant discussions as with the subexpression $G.id$.

$$\begin{aligned} \Delta_G(G) \cdot G + G \cdot \Delta_G(G) + \Delta_G(G) \cdot \Delta_G(G) + \Delta_G(G) \cdot (\mathbf{id}) + G \cdot \Delta_G(\mathbf{id}) \\ + \Delta_G(G) \cdot \Delta_G(\mathbf{id}) + \Delta_G(\mathbf{id}) \end{aligned}$$

Moreover, there are Δ_G operators on expressions that do not contain any G bindings, which can be further reduced to **zeros** using the DELTA-VAR-NEQ rule. Also, using the DELTA-VAR-EQ rule, all the $\Delta_G(G)$ expressions are reduced to delta variable instances δ_G . This yields the expression:

$$\delta_G \cdot G + G \cdot \delta_G + \delta_G \cdot \delta_G + \delta_G \cdot \mathbf{zeros} + G \cdot \mathbf{zeros} + \delta_G \cdot \mathbf{zeros} + \mathbf{zeros}$$

Fig. 5.8 demonstrates the delta rules for each of the core language constructs. The derivation of these rules are based on matrix identities. DELTA-ADD distributes the Δ operator across the summands. DELTA-MULT is directly derived from the distributivity of matrix multiplication over addition. DELTA-TRANS pushes the Δ into the expression before evaluating the transpose. DELTA-INV depicts the actual definition of Δ computation, which does not provide any computational savings at first glance, however it enables the Woodbury formula optimization that admits efficient evaluation. This is explained further in the example of section 5.4.1. DELTA-VAR-EQ simply maps the Δ_x of a matrix y to a variable instance (called the delta variable) if $x = y$, i.e., the matrix being changed x is identical to expression y . Similarly for DELTA-VAR-NEQ, if $x \neq y$, i.e., the matrix being changed x is different than the expression y , then the delta expression for y is **zeros**. DELTA-STACK distributes the Δ across the stacked matrices. DELTA-LET simply instantiates a delta variable instance and pushes the Δ across the expressions. Finally, DELTA-VECT and DELTA-DIAG reduce the Δ of the constant matrices to **zeros**.

Simplification Rules. The second class of reduction rules represents expression simplification. Symbolic computation is commonly accompanied by simplification. The derived expression is usually unnecessarily large and contains redundant computations. The expression is generally amenable to simplification. This is a major step in performing symbolic computations in computer algebra systems (CAS). For example, in CASs, right after deriving symbolic differentials, they usually perform simplification with the goal of minimizing the expression size. The same artifact happens while deriving Δ expressions, however, the goal is to avoid unnecessary redundant operations that will most probably result in higher cost. Fig. A.1 demonstrates a subset of simplification rules used within Lago. These kinds of transformation rules are important when the expression tree is undergoing derivation or major transformations by Lago and requires simplification along the way. For instance, consider the previous running example, there are many **zeros** matrices that have been created throughout the derivation process. After applying several simplification rules as demonstrated in Fig. A.1, our running example is simplified to the following expression:

$$\delta_G \cdot G + G \cdot \delta_G + \delta_G \cdot \delta_G$$

Note that simplification rules are always deterministically applied, and the choice of applying them is *not* left to the searching algorithm. To be more precise, simplification rules can be safely applied whenever possible and they do not increase the size of the search space.

Equivalence Rules

These are rules that define equivalent expressions $\text{lhs} \leftrightarrow \text{rhs}$. In particular, it is not clear beforehand which form of the expression (lhs or rhs) will result in an optimized program down the road. However, their presence is important not only because of their probable performance improvement, but also their possible impact on permitting other rewrite rules later. This effect is known as *enabling transformations* in compilers. Fig. A.2 presents a subset of equivalence rules used within Lago. Common subexpression elimination (CSE) and forward substitution (FS) are among these rules. In essence, these rules are the reverse of each other, hence it is not clear which one should be applied. General purpose compilers adopt CSE as a best-effort heuristic to enhance performance. That is, they apply them whenever possible as an enabling compiler optimization. Moreover, other domain-specific frameworks such as SystemML adopt these optimizations as static optimization opportunities, i.e., heuristics. In contrast, we argue that decisions about these optimizations should be taken under the light of cost-based optimization. In particular, algebraic and domain structure information often enable optimizations that override these general compiler heuristics. To illustrate this, consider our running example: $\delta_G \cdot G + G \cdot \delta_G + \delta_G \cdot \delta_G$, where the matrix G has dimensions $n \times n$. Now, suppose that δ_G is a simple single entry change which can be represented as an outer-product $u \cdot v^T$, i.e., $(n \times 1) \times (1 \times n)$. Using simple heuristics a compiler can directly detect that the expression $\delta_G = u \cdot v^T$ occurs several times within the program, hence by applying CSE, one can compute $u \cdot v^T$ once and then reuse it later on for further computation. In particular, the derived program becomes

$$\text{let } D := u \cdot v^T \text{ in } D \cdot G + G \cdot D + D \cdot D$$

Although CSE saved us from computing $u \cdot v^T$ more than once, i.e., saving $\mathcal{O}(n^2)$ operations, it results in more costly computations further on, in particular, the $\mathcal{O}(n^3)$ matrix multiplications $G \cdot D$, $D \cdot G$, and $D \cdot D$. On the other hand, given that $u \cdot v^T$ is an outer product of two vectors, using cost-based optimization, it is much cheaper, i.e., $\mathcal{O}(n^2)$ overall, to avoid CSE and keep the computations inlined as follows:

$$u \cdot (v^T \cdot G) + (G \cdot u) \cdot v^T + u \cdot (v^T \cdot u) \cdot v^T$$

$$\begin{array}{c}
 \frac{\mathbf{m}_1 : \mathcal{D}_{(n,m)}, \mathcal{C}_{(c_1)} \quad \mathbf{m}_2 : \mathcal{D}_{(n,m)}, \mathcal{C}_{(c_2)}}{\mathbf{m}_1 + \mathbf{m}_2 : \mathcal{D}_{(n,m)}, \mathcal{C}_{(c_1+c_2+n \cdot m)}} \text{DC-ADD} \\
 \frac{\mathbf{m}_1 : \mathcal{D}_{(n,m)}, \mathcal{C}_{(c_1)} \quad \mathbf{m}_2 : \mathcal{D}_{(m,p)}, \mathcal{C}_{(c_2)}}{\mathbf{m}_1 \cdot \mathbf{m}_2 : \mathcal{D}_{(n,p)}, \mathcal{C}_{(c_1+c_2+n \cdot m \cdot p)}} \text{DC-MULT} \\
 \frac{\mathbf{m} : \mathcal{D}_{(n,m)}, \mathcal{C}_{(c)}}{\mathbf{m}^\top : \mathcal{D}_{(m,n)}, \mathcal{C}_{(c+n \cdot m)}} \text{DC-TRANS} \\
 \frac{\mathbf{m}_1 : \mathcal{D}_{(n,m)}, \mathcal{C}_{(c_1)} \quad \mathbf{x} : \mathcal{D}_{(n,m)} \vdash \mathbf{m}_2 : \mathcal{D}_{(p,k)}, \mathcal{C}_{(c_2)}}{\mathbf{let } \mathbf{x} = \mathbf{m}_1 \mathbf{ in } \mathbf{m}_2 : \mathcal{D}_{(p,k)}, \mathcal{C}_{(c_1+c_2)}} \text{DC-LET} \\
 \frac{\mathbf{m}_1 : \mathcal{D}_{(n,m)}, \mathcal{C}_{(c_1)} \quad \mathbf{m}_2 : \mathcal{D}_{(n,p)}, \mathcal{C}_{(c_2)}}{\mathbf{m}_1 \Rightarrow \mathbf{m}_2 : \mathcal{D}_{(n,m+p)}, \mathcal{C}_{(c_1+c_2+(n \cdot (m+p)))}} \text{DC-STACK} \\
 \frac{}{\mathbf{vect}[c](s) : \mathcal{D}_{(1,c)}, \mathcal{C}_{(c)}} \text{DC-VECT} \\
 \frac{}{\mathbf{diag}[c](s) : \mathcal{D}_{(c,c)}, \mathcal{C}_{(c \cdot c)}} \text{DC-DIAG}
 \end{array}$$

Figure 5.9 – Inferring dimensions and cost of matrices.

This pattern occurs frequently in the derivation of incremental programs as we will demonstrate later in the examples of section 5.4. Equivalence rewrite rules construct programs which should be included in the search space. This is because it is not possible to decide locally if a rewrite rule will produce a better program or not. Even if it locally generates a better program, it might disable further transformations along the way. In other words, in order to not fall into a local optimum, one should traverse the search space of equivalent programs and rely on the search algorithm along with cost estimation to decide globally which program to pick.

Some transformation rules require specific conditions to check for their applicability in order to preserve semantics. These are known as *conditional rewrite rules* in the literature [104]. Apart from the structure of the program, these rules can use meta-information to check their applicability. This way, the framework can make sure that the transformation rules are *sound*, meaning that they do not change the program semantics. For example, \mathbf{m}^\top is equivalent to \mathbf{m} in the case that the matrix \mathbf{m} is symmetric. Next, we discuss meta-information and how it is inferred.

5.3.4 Meta-Information

DBMSs extensively use workload-specific information such as selectivity and cardinality in order to estimate query costs during query optimization. We observe that the idea of *meta-information* used by query optimizers can be abstracted and used for other domains. For example, in the domain of matrix algebra, symmetry, dimensions, structure, and rank of matrices are workload-specific information that permits further enhancements. To that end,

Lago permits encoding information about matrix and expression properties. The data type is the most obvious example of such information, e.g., a matrix of boolean elements or a matrix of double-precision numbers, e.g., Fig. A.3. Matrix dimensions are another, which are, in essence, very similar to relation cardinalities. The sparsity and the rank of a matrix are similar to the notion of selectivity in databases. Finally, the cost estimate itself can also be considered as another type of meta-information that can be used during cost-based optimization.

There are many benefits to meta-information: 1) First, to verify program correctness. For example, in the case of matrix multiplication, dimensions are used to check if the number of columns of the first matrix is equal to the number of rows of the second matrix. 2) Secondly, to guide the optimizing compiler to reason about optimization opportunities by evaluating cost estimates. 3) Thirdly, to enable conditional rewrites. 4) Finally, to enable further specialization opportunities during the code generation phase.

Lago is extensible; in a sense, one can introduce more properties that capture the workload-specific information available. These properties are not limited to the information about the input data provided by the input program. Similar to type inference algorithms, Lago tries to propagate this meta-information throughout the whole program whenever possible. This is achieved by defining meta-information inference rules as described next.

Inference Rules

The user provides the information about the input matrices by specifying their associated meta-information. In order to leverage this information, they should be propagated throughout the program. This way, the optimizing compiler can utilize the provided information for the whole program. The Lago framework requires inference rules in order to infer the information of an expression based on its input data through a bottom-up derivation approach. These rules are similar to type inference rules which are used in type systems of programming languages. For instance, Fig. 5.9 illustrates the inference rules for matrix dimensions. Fig. A.3, Fig. A.6, Fig. A.5, and Fig. A.4 in the appendix demonstrate a subset of the inference rules for matrix symmetry, triangular matrices, matrix rank, and diagonal matrices respectively.

For example, consider our original reachability example. If the input graph G is undirected, then it is symmetric and only requires to be stored in a lower triangular binary adjacency matrix. As depicted in Fig. 5.10, starting from these properties of matrix G , the information propagates upwards to infer the meta-information of the intermediate and final results using the inference rules described before. This information is useful for specialization purposes. For instance, boolean matrices can be represented in a more compact form (e.g., bit vectors) in comparison to the more general double-precision matrices. Similarly, lower-triangular matrices can be represented using less space given the symmetric nature of the matrix. Moreover, specialized

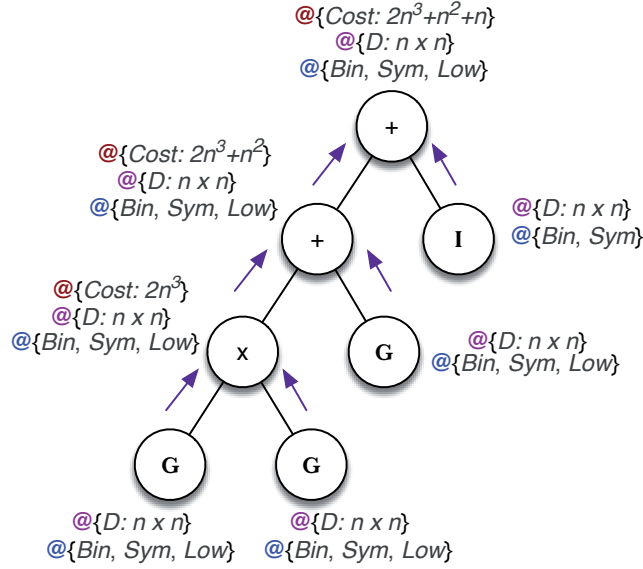


Figure 5.10 – Meta-Information propagating bottom-up using their respective inference rules. Bin, Sym, Low, D, and Cost correspond to the inferred Binary, Symmetric, Lower-Triangular, Dimensions, and Cost properties respectively.

implementations of operations can be leveraged using the knowledge of symmetry. Similar inference rules can be applied during the evaluation of the trigger program, e.g., $G \cdot U \cdot V$ is symmetric if $U \cdot V$ is symmetric.

Cost Model. One of the most essential meta-information instances is cost estimation. The cost estimate is used to guide the search space in choosing efficient derived programs. Currently, the cost estimate is modelled as a function of the number of arithmetic operations that need evaluation. Similar to cost estimation in database systems which requires cardinality information of relations, the cost estimation in Lago also requires knowledge about the dimensions of matrices. This means, before starting the inference process for the cost estimation meta-information, the dimension inference should be performed. The inference rules for cost estimation are given in Fig. 5.9. Cost and dimensions inference are illustrated in Fig. 5.10. Returning back to our running example, these inference rules helped the search algorithm in estimating the cost and favoring forward substitution of δ_G over CSE given the following

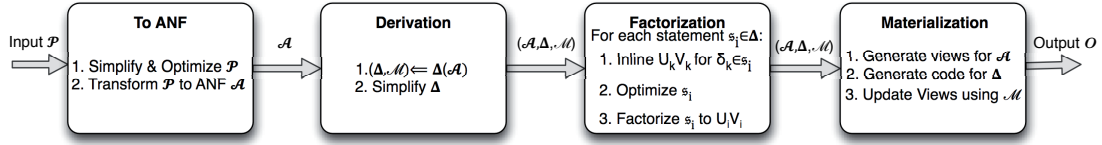


Figure 5.11 – Lago IVM phases.

parenthesization order:

$$u \cdot (v^T \cdot G) + (G \cdot u) \cdot v^T + u \cdot (v^T \cdot u) \cdot v^T$$

Cost estimation is extensible as well. For instance, in addition to dimensions, one can introduce and define additional meta-information abstractions that can introduce specialized solutions and, therefore, better cost estimation. For example, based on the structure of matrices, e.g., upper/lower triangular, one can use specialized matrix multiplication algorithms, e.g., SSYMM in BLAS, that only computes the required entries, thereby saving space and computation cost. This can be reflected in the cost model by inferring the structure of the matrix, e.g., Fig. A.6, before performing cost inference. By further reflecting this information as new inference rules in the cost estimation, one could further specialize the precision of the cost model estimation.

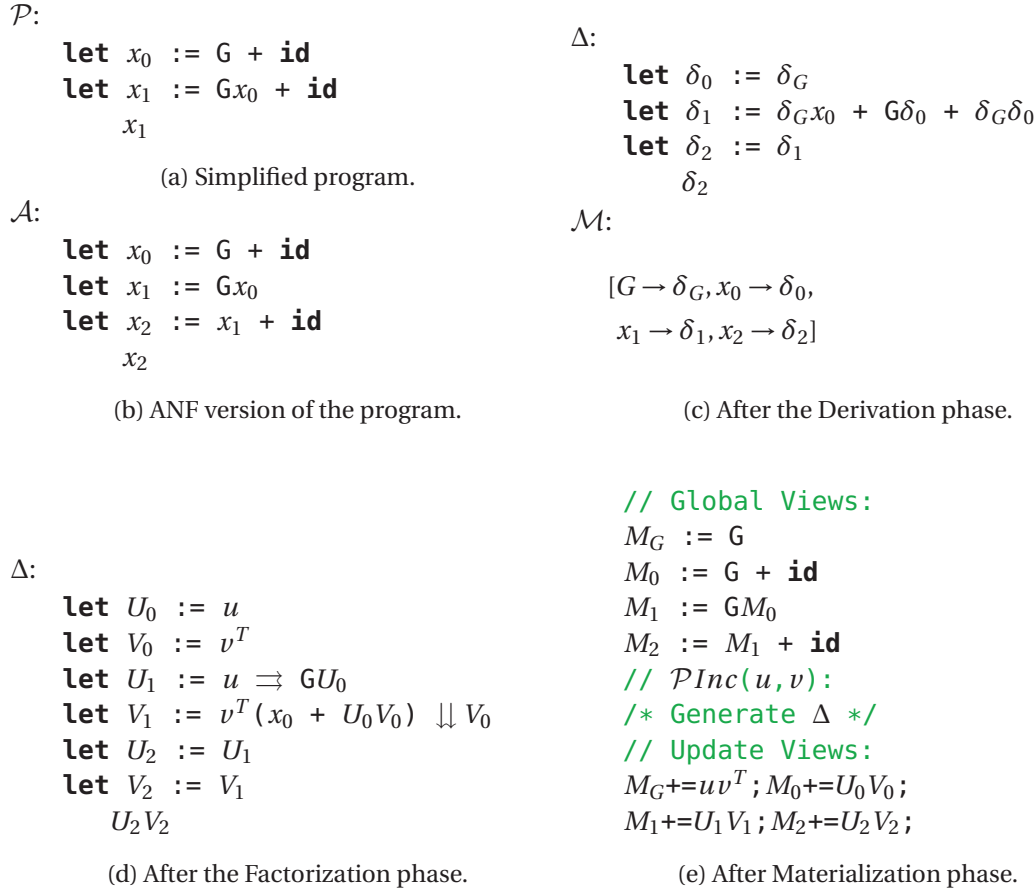
5.3.5 Wiring it all together

In the previous sections, we have discussed the building blocks that constitute the Lago framework. In this section, we discuss how all these parts are put together to generate incremental trigger programs. An incremental program consists of an initialization phase that precomputes and materializes the initial value of the intermediate and result expressions, and a trigger function that computes a set of Δ expressions that update their corresponding views^V.

To achieve these goals, an input program accepted by Lago undergoes several phases. As depicted in Fig. 5.11, the input program passes through four stages, namely, ANF, DERIVATION, FACTORIZATION and MATERIALIZATION. Next, we explain each phase while illustrating it using our running example.

1. ANF. As a preprocessing step, an input Lago program \mathcal{P} is first simplified and then optimized using cost-based optimization to find an appropriate ordering of operations. After that, it is converted to the administrative normal form (ANF) \mathcal{A} [76]. In our context, the ANF is defined as a simple representation of the program that assigns a unique variable to each subexpression

^VFor presentation clarity purposes, in the rest of this chapter we omit the “.” symbol whenever multiplication is understood within context.


 Figure 5.12 – Going through the IVM phases of program \mathcal{P} from Fig. 5.7.

while also ensuring that each variable is assigned before it is used. The ANF is extensively used in optimizing compilers due to its simplistic canonical representation that facilitates reasoning and optimization [76]. To explain this, consider our reachability program \mathcal{P} as depicted in Fig. 5.7. First, the program is simplified using the simplification rules in Fig. A.1 where the **iterate** construct is unfolded and multiplications with the identity matrix are omitted, yielding the simple program in Fig 5.12a. Afterwards, in Fig. 5.12b, the simplified program is transformed into its ANF \mathcal{A} , where each simple subexpression containing one operation is assigned to a unique variable.

2. Derivation. In this phase, the delta derivation rules are recursively applied over the \mathcal{A} program reducing it to Δ . During derivation, a map \mathcal{M} is created that maps each intermediate result variable x_i in \mathcal{A} to its corresponding delta δ_i variable. Moreover, simplification rules are applied whenever possible. This ensures that each statement $\mathbf{s}_i \in \Delta$ represents a sum of matrix

products, i.e., $\Sigma_i m_i$ where m_i is an expression of matrix products. Fig. 5.12c presents the final delta program derived from $\Delta_G(\mathcal{A})$ and the corresponding map \mathcal{M} .

3. Factorization. The main goal of this phase is to represent each δ_i variable in a compact factored form $U_i V_i$. This is achieved by recursively propagating and forward substituting each δ_k variable with its corresponding factored form $U_k V_k$ within each statement that calls δ_k . Note that forward substitution begins with the initial substitution of δ_G with uv^T . Given that Δ is in ANF form, then it is ensured that δ_k and therefore, $U_k V_k$ is defined before being used. Then, each statement s_i is optimized using the equivalence rules along with cost-based optimizations. In particular, the search algorithm explores the search space created by applying valid equivalence rules on the statement s_i . Then, it chooses the program with minimum inferred cost as explained in section 5.3.4. Notice that search is confined within the scope of a single statement s_i . This guided approach avoids searching a vast search space that includes all the statements of the whole program.

This ensures that each statement $s_i \in \Delta$ is a sum of matrix products containing $U_k V_k$, i.e., $\Sigma_j p_j q_j$ where p_j and q_j are expressions of matrix products and j represents the index of minimum dimension within the overall matrix products. Finally, each statement is factorized to $U_i V_i$ using the FACTORIZATION rule in Fig. A.2 (Appendix), such that $U_i = p_0 \Rightarrow p_1 \cdots \Rightarrow p_n$ and similarly $V_i = q_0 \Downarrow q_1 \cdots \Downarrow q_n$. Fig. 5.12d presents the factorized Δ program with its corresponding updated map \mathcal{M} .

4. Materialization. Finally, in this phase Lago generates the incremental program. First, it generates global materialized views for each of the variables defined in \mathcal{A} . Then, it generates the trigger program Δ derived from the previous stage. Finally, it updates the global views with their corresponding δ_i expression derived in \mathcal{M} . Fig. 5.12e illustrates the final incremental program for P given incremental changes to G , i.e., $\delta_G := uv^T$.

5.4 Other Use cases

To further illustrate the stages undertaken by Lago, we demonstrate how it automatically generates trigger programs for several use case examples other than our running graph example. Moreover, the performance results for the IVM of these use cases can be found in the evaluation section.

5.4.1 Incremental Linear Regression

Linear regression is an approach for modelling the relationship between the dependent variables $Y_{m \times j}$ and independent variables $X_{m \times n}$. It is extensively used in fitting predictive models to an observed dataset of X and Y values. The goal is to estimate, given the input, the unknown

parameters. The ordinary least squares method solves this problem by finding a statistical estimate of the parameter β^* best satisfying $Y = X\beta$. The program, written as a linear algebra program, is $\beta^* := (X^T X)^{-1} X^T Y$. The evaluation of the previous closed form equation requires expensive $\mathcal{O}(n^3)$ matrix-matrix computations for computing matrix multiplications and inverses. It is far from efficient to re-evaluate the expression over and over again as the input datasets evolve. Input datasets naturally evolve by either growing (for example as more observations are accumulated to X and Y), or by changing (as more accurate estimates arrive). Alternatively, Lago derives materialized views of precomputed intermediate results and their corresponding delta expressions to generate trigger programs.

Here, we focus on how to derive the delta expression for β^* under updates to X with $\delta_X := uv^T$. Fig. 5.13 demonstrates the derivation of the incremental program going through the IVM phases. Fig. 5.13a presents the the original Lago program. After the program is simplified, optimized and converted to the ANF representation \mathcal{A} , we compute its delta with an initial \mathcal{M} of δ_X as depicted in Fig. 5.13b. Afterwards, the DERIVATION PROCESS starts where several delta derivation steps and simplifications are applied as illustrated in Fig. 5.13c to Fig. 5.13g. In Fig. 5.13h, the FACTORIZATION phase starts by forward substituting the δ_X variable (defined in \mathcal{M}) with uv^T . The new delta expression for δ_0 is then derived. After applying a few simplifications, Lago factorizes δ_0 into $U_0 V_0$ as demonstrated in Fig. 5.13j and Fig. 5.13k, which in turn is inlined into all variable calls of δ_0 while updating \mathcal{M} simultaneously. The same approach is applied recursively, finally yielding, in Fig. 5.13o, all the delta expressions in factored form and their respective materialized views defined in \mathcal{M} . In the end, during the materialization phase, Lago uses the derived \mathcal{A} to generate the global views for the intermediate results $x_0 := X^T X$, $x_1 := x_0^{-1}$, and $x_2 := X^T Y$ and for the final program $x_3 := x_1 x_2$. Additionally, it generates the trigger program using the derived delta expressions for each materialized view represented in $U_0 V_0$, $U_1 V_1$, $U_2 V_2$, and $U_3 V_3$ respectively.

5.4.2 Incremental Matrix Powers

Matrix powers play an important role in many different domains including evaluating the stochastic matrix of a Markov chain after k steps and solving systems of linear differential equations using matrix exponentials. They also lay the foundation for more advanced analytics like batch gradient descent and furthermore, computing graph analytics.

Consider the same running example as in section 5.2 that computes the 8^{th} power of an input matrix A as depicted in Fig 5.14. The original program can be represented using a simple **iterate** construct as demonstrated in Fig 5.14a. Once again, the evaluation of the program requires expensive $\mathcal{O}(n^3)$ matrix-matrix computations. Re-evaluation of the entire program on any delta change $\delta_A := uv^T$ is a costly process. On the other hand, Lago derives the delta of these expressions on each incremental change. First, the program is converted to ANF in

Fig. 5.14b. Then, the DERIVATION phase starts by applying delta derivation rules as depicted in Fig. 5.14c to Fig. 5.14g while updating \mathcal{M} to include the mappings between intermediate result variables and their corresponding delta variables. In Fig. 5.14h, the FACTORIZATION phase starts by replacing δ_A with uv^T which is inlined into the expression. Thereby, the δ_0 variable is evaluated and optimized using the appropriate equivalence rules generating the corresponding factorized form $U_0 V_0$ in Fig. 5.14k. The same process is repeated in a recursive manner evaluating the delta expressions $U_1 V_1$, $U_2 V_2$, and $U_3 V_3$ for each of x_0 , x_1 , x_2 and the final program x_3 .

```
/*Original program*/
 $(X^T X)^{-1} X^T Y$ 
```

(a)

```
/*Transform to ANF then
compute Delta of it*/
```

```
 $\Delta_X(\text{let } x_0 := (X^T X) \text{ in}$ 
 $\text{let } x_1 := x_0^{-1} \text{ in}$ 
 $\text{let } x_2 := X^T Y \text{ in}$ 
 $\text{let } x_3 := x_1 x_2 \text{ in } x_3)$ 
```

(b) $\mathcal{M} = [X \rightarrow \delta_X]$

```
/*Apply Delta-let rule*/
```

```
 $\text{let } \delta_0 := \Delta_X((X^T X)) \text{ in}$ 
 $\Delta_X(\text{let } x_1 := x_0^{-1} \text{ in}$ 
 $\text{let } x_2 := X^T Y \text{ in}$ 
 $\text{let } x_3 := x_1 x_2 \text{ in } x_3)$ 
```

(c) $\mathcal{M} = [X \rightarrow \delta_X, x_0 \rightarrow \delta_0]$

```
/*Apply Delta-Mult rule*/
```

```
 $\text{let } \delta_0 := \delta_X^T X +$ 
 $X^T \delta_X + \delta_X^T \delta_X \text{ in}$ 
 $\Delta_X(\text{let } x_1 := x_0^{-1} \text{ in}$ 
 $\text{let } x_2 := X^T Y \text{ in}$ 
 $\text{let } x_3 := x_1 x_2 \text{ in } x_3)$ 
```

(d) $\mathcal{M} = [X \rightarrow \delta_X, x_0 \rightarrow \delta_0]$

```
/*Apply delta rules*/
```

```
 $\text{let } \delta_0 := \delta_X^T X + X^T \delta_X + \delta_X^T \delta_X \text{ in}$ 
 $\text{let } \delta_1 := (x_0 + \delta_0)^{-1} - x_0^{-1} \text{ in}$ 
 $\text{let } \delta_2 := \delta_X^T Y + X^T \Delta_X(Y) + \delta_X^T \Delta_X$ 
 $(Y) \text{ in}$ 
 $\text{let } \delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2 \text{ in } \delta_3$ 
```

(e) $\mathcal{M} = [X \rightarrow \delta_X, x_0 \rightarrow \delta_0$
 $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$

```
/*Apply Delta-Ind-Var rule*/
```

```
 $\text{let } \delta_0 := \delta_X^T X + X^T \delta_X + \delta_X^T \delta_X \text{ in}$ 
 $\text{let } \delta_1 := (x_0 + \delta_0)^{-1} - x_0^{-1} \text{ in}$ 
 $\text{let } \delta_2 := \delta_X^T Y + X^T \text{zeros} + \delta_X^T \text{zeros}$ 
 $\text{in}$ 
 $\text{let } \delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2 \text{ in } \delta_3$ 
```

(f) $\mathcal{M} = [X \rightarrow \delta_X, x_0 \rightarrow \delta_0$
 $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$

```
/*perform simplifications*/
```

```
 $\text{let } \delta_0 := \delta_X^T X + X^T \delta_X + \delta_X^T \delta_X \text{ in}$ 
 $\text{let } \delta_1 := (x_0 + \delta_0)^{-1} - x_0^{-1} \text{ in}$ 
 $\text{let } \delta_2 := \delta_X^T Y \text{ in}$ 
 $\text{let } \delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2 \text{ in } \delta_3$ 
```

(g) $\mathcal{M} = [X \rightarrow \delta_X, x_0 \rightarrow \delta_0$
 $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$

```
/*Begin: Factorization Phase
```

```
 $\text{inline } \delta_X := uv^T \text{ rule*/}$ 
 $\text{let } \delta_0 := (uv^T)^T X +$ 
 $X^T uv^T + (uv^T)^T uv^T \text{ in}$ 
 $\text{let } \delta_1 := (x_0 + \delta_0)^{-1} - x_0^{-1} \text{ in}$ 
 $\text{let } \delta_2 := (uv^T)^T Y \text{ in}$ 
 $\text{let } \delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2 \text{ in } \delta_3$ 
```

(h) $\mathcal{M} = [X \rightarrow uv^T, x_0 \rightarrow \delta_0$
 $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$

Figure 5.13 – Step-by-step Δ derivation of the Ordinary Least Squares program till the factorization phase.

```
/*Transpose homomorphism*/
```

```
let  $\delta_0 := v u^T X +$   
 $X^T u v^T + v u^T u v^T$  in  
let  $\delta_1 := (x_0 + \delta_0)^{-1} - x_0^{-1}$  in  
let  $\delta_2 := (v u^T) Y$  in  
let  $\delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2$  in  $\delta_3$ 
```

$$(i) \quad \mathcal{M} = [X \rightarrow u v^T, x_0 \rightarrow \delta_0 \\ x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$$

```
/*factorize and replace  $\delta_0$ */
```

```
let  $U_0 := v \Rightarrow X^T u$  in  
let  $V_0 := u^T (X + u v^T) \Downarrow v^T$  in  
let  $\delta_1 := (x_0 + \delta_0)^{-1} - x_0^{-1}$  in  
let  $\delta_2 := (v u^T) Y$  in  
let  $\delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2$  in  $\delta_3$ 
```

$$(k) \quad \mathcal{M} = [X \rightarrow u v^T, x_0 \rightarrow U_0 V_0 \\ x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$$

```
/*Apply Woodbury formula*/
```

```
let  $U_0 := v \Rightarrow X^T u$  in  
let  $V_0 := u^T (X + u v^T) \Downarrow v^T$  in  
let  $\delta_1$   
   $:= -x_0^{-1} U_0 (\text{id} + V_0 x_0^{-1} U_0)^{-1} V_0 x_0^{-1}$  in  
let  $\delta_2 := (v u^T) Y$  in  
let  $\delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2$  in  $\delta_3$ 
```

$$(m) \quad \mathcal{M} = [X \rightarrow u v^T, x_0 \rightarrow U_0 V_0 \\ x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$$

```
/*Fast Forward:Repeating  
previous rules*/
```

```
let  $U_0 := v \Rightarrow X^T u$  in  
let  $V_0 := u^T (X + u v^T) \Downarrow v^T$  in  
let  $U_1 := -x_1 U_0$  in  
let  $V_1 := (\text{id} + V_0 x_0^{-1} U_0)^{-1} V_0 x_0^{-1}$  in  
let  $U_2 := v$  in let  $V_2 = u^T Y$  in  
let  $U_3 := U_1 \Rightarrow x_1 U_2$  in  
let  $V_3 := V_1 (x_2 + U_2 V_2) \Downarrow V_2$  in  
 $U_3 V_3$ 
```

$$90 \quad (o) \quad \mathcal{M} = [X \rightarrow u v^T, x_0 \rightarrow U_0 V_0 \\ x_1 \rightarrow U_1 V_1, x_2 \rightarrow U_2 V_2, x_3 \rightarrow U_3 V_3]$$

```
/*cost based factorization*/
```

```
let  $\delta_0 := (v \Rightarrow X^T u) (u^T (X + u v^T)$   
   $) \Downarrow v^T$  in  
let  $\delta_1 := (x_0 + \delta_0)^{-1} - x_0^{-1}$  in  
let  $\delta_2 := (v u^T) Y$  in  
let  $\delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2$  in  $\delta_3$ 
```

$$(j) \quad \mathcal{M} = [X \rightarrow u v^T, x_0 \rightarrow \delta_0 \\ x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$$

```
/*inline  $\delta_0 = U_0 V_0$  rule*/
```

```
let  $U_0 := v \Rightarrow X^T u$  in  
let  $V_0 := u^T (X + u v^T) \Downarrow v^T$  in  
let  $\delta_1 := (x_0 + U_0 V_0)^{-1} - x_0^{-1}$  in  
let  $\delta_2 := (v u^T) Y$  in  
let  $\delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2$  in  $\delta_3$ 
```

$$(l) \quad \mathcal{M} = [X \rightarrow u v^T, x_0 \rightarrow U_0 V_0 \\ x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$$

```
/*Fast Forward:Repeating  
previous rules*/
```

```
let  $U_0 := v \Rightarrow X^T u$  in  
let  $V_0 := u^T (X + u v^T) \Downarrow v^T$  in  
let  $U_1 := -x_1 U_0$  in  
let  $V_1 := (\text{id} + V_0 x_0^{-1} U_0)^{-1} V_0 x_0^{-1}$  in  
let  $\delta_2 := (v u^T) Y$  in  
let  $\delta_3 := \delta_1 x_2 + x_1 \delta_2 + \delta_1 \delta_2$  in  $\delta_3$ 
```

$$(n) \quad \mathcal{M} = [X \rightarrow u v^T, x_0 \rightarrow U_0 V_0 \\ x_1 \rightarrow U_1 V_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$$

```
/*Original program*/
iterate[4](A)(acc=>
acc · acc)
```

(a)

```
/*Apply iterate unfold rule &
Transform into ANF
then compute Delta*/
```

```
 $\Delta_A$ ( let  $x_0 := AA$  in
  iterate[3]( $x_0$ )(acc=>
acc · acc))
```

(b) $\mathcal{M} = [A \rightarrow \delta_A]$

```
/*Recurse iterate unfold rule*/
```

```
 $\Delta_A$ (let  $x_0 := AA$  in
let  $x_1 := x_0 x_0$  in
let  $x_2 := x_1 x_1$  in
let  $x_3 := x_2 x_2$  in
iterate[0]( $x_3$ )(acc=>
acc · acc))
```

(c) $\mathcal{M} = [A \rightarrow \delta_A]$

```
/*Apply simplification rule*/
```

```
 $\Delta_A$ (let  $x_0 := AA$  in
let  $x_1 := x_0 x_0$  in
let  $x_2 := x_1 x_1$  in
let  $x_3 := x_2 x_2$  in
 $x_3$ )
```

(d) $\mathcal{M} = [A \rightarrow \delta_A]$

```
/*Apply delta-let rule*/
```

```
let  $\delta_0 := \Delta(AA)$  in
 $\Delta_A$ (let  $x_1 := x_0 x_0$  in
let  $x_2 := x_1 x_1$  in
let  $x_3 := x_2 x_2$  in
 $x_3$ )
```

(e) $\mathcal{M} = [A \rightarrow (A, \delta_A), x_0 \rightarrow (AA, \delta_0)]$

```
/*Recurse iterate unfold rule*/
```

```
let  $\delta_0 := \delta_A A + A \delta_A + \delta_A \delta_A$  in
let  $\delta_1 := \delta_0 x_0 + x_0 \delta_0 + \delta_0 \delta_0$  in
let  $\delta_2 := \delta_1 x_1 + x_1 \delta_1 + \delta_1 \delta_1$  in
let  $\delta_3 := \delta_2 x_2 + x_2 \delta_2 + \delta_2 \delta_2$  in
 $\Delta_A(x_3)$ 
```

(f) $\mathcal{M} = [A \rightarrow \delta_A, x_0 \rightarrow \delta_0$
 $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$

```
/*Apply delta rule*/
```

```
let  $\delta_0 := \delta_A A + A \delta_A + \delta_A \delta_A$  in
let  $\delta_1 := \delta_0 x_0 + x_0 \delta_0 + \delta_0 \delta_0$  in
let  $\delta_2 := \delta_1 x_1 + x_1 \delta_1 + \delta_1 \delta_1$  in
let  $\delta_3 := \delta_2 x_2 + x_2 \delta_2 + \delta_2 \delta_2$  in
 $\delta_3$ 
```

(g) $\mathcal{M} = [A \rightarrow \delta_A, x_0 \rightarrow \delta_0$
 $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$

```
/*Begin Factorization phase
```

```
inline  $\delta_A := uv^T$  rule*/
let  $\delta_0 := uv^T A + A uv^T + uv^T uv^T$ 
in
let  $\delta_1 := \delta_0 x_0 + x_0 \delta_0 + \delta_0 \delta_0$  in
let  $\delta_2 := \delta_1 x_1 + x_1 \delta_1 + \delta_1 \delta_1$  in
let  $\delta_3 := \delta_2 x_2 + x_2 \delta_2 + \delta_2 \delta_2$  in
 $\delta_3$ 
```

(h) $\mathcal{M} = [A \rightarrow uv^T, x_0 \rightarrow \delta_0$
 $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$

```

/*cost based optimization*/
let  $\delta_0 := uv^T A + (Au + uv^T u)v^T$  in
let  $\delta_1 := \delta_0 x_0 + x_0 \delta_0 + \delta_0 \delta_0$  in
let  $\delta_2 := \delta_1 x_1 + x_1 \delta_1 + \delta_1 \delta_1$  in
let  $\delta_3 := \delta_2 x_2 + x_2 \delta_2 + \delta_2 \delta_2$  in
 $\delta_3$ 

(i)  $\mathcal{M} = [A \rightarrow uv^T, x_0 \rightarrow \delta_0$ 
       $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$ 

/*Factorize  $\delta_0$ */
let  $U_0 := u \Rightarrow (Au + uv^T u)$  in
let  $V_0 := v^T A \Downarrow v^T$  in
let  $\delta_0 := U_0 V_0$  in
let  $\delta_1 := \delta_0 x_0 + x_0 \delta_0 + \delta_0 \delta_0$  in
let  $\delta_2 := \delta_1 x_1 + x_1 \delta_1 + \delta_1 \delta_1$  in
let  $\delta_3 := \delta_2 x_2 + x_2 \delta_2 + \delta_2 \delta_2$  in
 $\delta_3$ 

(k)  $\mathcal{M} = [A \rightarrow uv^T, x_0 \rightarrow \delta_0$ 
       $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$ 

/*Cost based optimization*/
let  $U_0 := u \Rightarrow (Au + uv^T u)$  in
let  $V_0 := v^T A \Downarrow v^T$  in
let  $\delta_1 := U_0 \Rightarrow (x_0 U_0 + U_0 V_0 U_0)$ 
       $V_0 x_0 \Downarrow V_0$  in
let  $\delta_2 := \delta_1 x_1 + x_1 \delta_1 + \delta_1 \delta_1$  in
let  $\delta_3 := \delta_2 x_2 + x_2 \delta_2 + \delta_2 \delta_2$  in
 $\delta_3$ 

(m)  $\mathcal{M} = [A \rightarrow uv^T, x_0 \rightarrow U_0 V_0$ 
       $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$ 

/*cost based factorization*/
let  $\delta_0 := (u \Rightarrow Au + uv^T u) (v^T A \Downarrow$ 
       $v^T)$  in
let  $\delta_1 := \delta_0 x_0 + x_0 \delta_0 + \delta_0 \delta_0$  in
let  $\delta_2 := \delta_1 x_1 + x_1 \delta_1 + \delta_1 \delta_1$  in
let  $\delta_3 := \delta_2 x_2 + x_2 \delta_2 + \delta_2 \delta_2$  in
 $\delta_3$ 

(j)  $\mathcal{M} = [A \rightarrow uv^T, x_0 \rightarrow \delta_0$ 
       $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$ 

/*inline  $\delta_0 = UV$  rule*/
let  $U_0 := u \Rightarrow (Au + uv^T u)$  in
let  $V_0 := v^T A \Downarrow v^T$  in
let  $\delta_1 := U_0 V_0 x_0 + x_0 U_0 V_0 +$ 
       $U_0 V_0 U_0 V_0$  in
let  $\delta_2 := \delta_1 x_1 + x_1 \delta_1 + \delta_1 \delta_1$  in
let  $\delta_3 := \delta_2 x_2 + x_2 \delta_2 + \delta_2 \delta_2$  in
 $\delta_3$ 

(l)  $\mathcal{M} = [A \rightarrow uv^T, x_0 \rightarrow U_0 V_0$ 
       $x_1 \rightarrow \delta_1, x_2 \rightarrow \delta_2, x_3 \rightarrow \delta_3]$ 

/*Fast Forward: Repeating
rules*/
let  $U_0 := u \Rightarrow (Au + uv^T u)$  in
let  $V_0 = v^T A \Downarrow v^T$  in
let  $U_1 := U_0 \Rightarrow (x_0 U_0 + U_0 V_0 U_0)$ 
      in let  $V_1 := V_0 x_0 \Downarrow V_0$  in
let  $U_2 := U_1 \Rightarrow (x_1 U_1 + U_1 V_1 U_1)$ 
      in let  $V_2 := V_1 x_1 \Downarrow V_1$  in
let  $U_3 := U_2 \Rightarrow (x_2 U_2 + U_2 V_2 U_2)$  in
let  $V_3 := V_2 x_2 \Downarrow V_2$  in
 $U_3 V_3$ 

(n)  $\mathcal{M} = [A \rightarrow uv^T, x_0 \rightarrow U_0 V_0$ 
       $x_1 \rightarrow U_1 V_1, x_2 \rightarrow U_2 V_2, x_3 \rightarrow U_3 V_3]$ 

```

Figure 5.14 – Step-by-step Δ derivation of Matrix powers till the factorization phase.

5.5 Related Work

This section presents related work in different directions.

Computer Algebra Systems. CAS is a software program that allows computation over mathematical expressions and that automates tedious and difficult algebraic manipulation tasks. They can perform symbolic computations including differentiation and integration. Examples include Mathematica [181], MAPLE [134] and additional packages in Theano [167]. Similarly, Lago performs symbolic computation in a sense that it derives Δ expressions using the reduction rules that we present in this chapter. Moreover, Lago differs from CAS in its ability to derive incremental programs; perform cost-based optimization; and generate efficient specialized code.

Scientific Databases. RasDaMan [24] and AML [126] represent database systems that are specialized in array processing. They provide infrastructure for expressing and optimizing queries over multidimensional arrays. Queries are translated into an array algebra and optimized using a large collection of transformation rules. ASAP [163] supports scientific computing primitives on a storage manager optimized for storing multidimensional arrays. Additionally, RIOT [192] provides an efficient out-of-core framework for scientific computing. However, none of these systems support incremental computation. In contrast, Lago is specialized for supporting IVM of matrix programs. Moreover, it provides a generic unified framework for different semiring configurations of matrix algebra.

High Performance Computing. There is high demand for efficient matrix manipulation in numerical and scientific computing. BLAS [68] exposes a set of low-level routines that represent common linear algebra primitives for higher-level libraries including LINPACK, LAPACK, and ScaLAPACK for parallel processing. Hardware vendors such as Intel or AMD and code generators such as ATLAS [179] provide highly optimized BLAS implementations for dense linear algebra. Moreover, other works such as Combinatorial BLAS [38, 66] provide efficient BLAS implementations dedicated for sparse linear algebra. All of this work is orthogonal to Lago as it operates at a higher level of abstraction. In essence, IVM translates input matrix programs to trigger code that calls cheaper matrix BLAS primitives.

Iterative Computation. Recently, there has been a growing interest in designing frameworks for iterative and incremental computation. The differential dataflow model [129] presents a new methodology to model incremental computation for iterative algorithms. Their approach relies on the assumption that input changes result in small changes down the road. However, this assumption does not hold for matrix algebra programs because of the avalanche effect of input changes as described in this chapter. For iterative applications under the MapReduce framework, several systems [37, 71, 191] have been proposed. They present techniques that cache and index loop-invariant data on local disks and persist materialized views between

iterations. Moreover, Dryad [100] and Spark [187] represent systems that support iterative computation under the general DAG execution model. Mahout, MLbase [109] and others [55, 174, 132] provide scalable machine learning and data mining tools. All these systems are orthogonal to Lago. Our work is concerned with the design and implementation of a compiler framework for the incremental view maintenance of matrix algebra. Moreover, the framework can be easily coupled with any of these underlying systems at the code generation layer as we illustrate in the evaluation section 5.6 with Spark.

IVM and Stream Processing. Incremental View Maintenance techniques [30, 107, 87] support incremental updates of database materialized views by employing differential algorithms to re-evaluate the view expression. Chirkova *et al.* [50] present a detailed survey on this direction. Moreover, data stream processing engines [7, 135, 21] incrementally evaluate continuous queries as windows advance over unbounded input streams. In contrast to all the previous approaches, this chapter targets incremental maintenance of linear algebra programs as opposed to classical database (SQL) queries. The linear algebra domain has different semantics and primitives; thus, the challenges and optimization techniques widely differ.

Graph Analytics. There is plethora of frameworks dedicated for graph processing including Powergraph [80], Pregel [124], GraphLab [119, 120], GraphChi [113], and Galois [139]. They provide various programming models specialized for graph processing based on Bulk Synchronous Programming. Recently, there has been work on representing graph algorithms using sparse matrix manipulation operations including CombBLAS [38], GraphMat [165], and Graphblas[66]. However, none of these systems support incremental computation. There have been several works that target incremental computation of specific graph problems [51, 84], including connectivity [94], minimum spanning tree [94], transitive closure [57, 58], and all-pairs shortest path [59, 106]. However, each of these solutions aim at a particular graph problem and are not represented as matrix computations. In contrast, Lago provides a general matrix framework that supports graph IVM, cost-based optimization, and low-level specializations.

Linear Algebra DSLs. The Spiral [149] project provides a domain-specific compiler for synthesizing digital signal processing kernels, e.g., Fourier transforms. The authors present the SPL [183] language that expresses recursion and formulas in a mathematical form. They present a framework that optimizes at the algorithmic and implementation level and that uses runtime information to guide the synthesis process. The LGen compiler [159] targets small scale basic linear algebra computations of fixed size linear algebra expressions which are common in graphics and media processing applications. The authors present two level DSLs, namely LL to perform tiling decisions and Σ -LL to enable loop level optimizations. The generated output is a C function that includes intrinsics to enable SIMD vector extensions. Orthogonally, Lago targets IVM of LA programs for different domains, i.e., semiring configurations, and is restricted to high-level optimizations. The closest to our work is the basic

linear algebra compiler presented in [74]. It decomposes a linear algebra target equation into a sequence of computations provided by BLAS or LAPACK and generates associated Matlab code. Similar to our work, their approach exploits domain knowledge and properties of the operands by rewriting and inference rules. However, we focus on IVM and optimization under this setting.

Incremental Statistical Frameworks. Bayesian inference [26] uses the Bayes' rule to update the hypothesis's probability estimate as additional evidence is acquired. A variety of applications can be built on top of these frameworks including pattern recognition and classification. Our work focuses on incrementalizing applications that can be expressed as linear algebra programs and generating efficient incremental programs for different runtime environments.

Programming Languages. The PL community has extensively explored the direction of incremental computation and information flow [47]. They have developed compilation techniques that translate high-level programs into executables that are amenable to dynamic changes. Moreover, self-adjusting computation supports incremental computation by exploiting dynamic dependency graphs and change propagation algorithms [9, 47]. However, these approaches differ from our work on several dimensions: *a)* Firstly, they target general purpose programs in comparison to our domain-specific approach. *b)* Secondly, they require developer knowledge and involvement by annotating the modifiable parts of the program. *c)* Finally, they cannot capture the propagation of deltas across statements and efficiently represent them in a compressed form as presented in this chapter.

5.6 Evaluation

In the previous sections, we have presented a concrete framework for expressing, deriving, and optimizing incremental view maintenance of matrix algebra programs. In this section, we demonstrate the performance of the derived incremental programs in comparison to re-evaluation. We illustrate two case studies that build upon matrix algebra: computing linear regression and evaluating graph reachability and shortest path after k hops. Moreover, we evaluate the opportunity benefits of specialization leveraged by inferred meta-information. We show how Lago pushes the burden and complications of IVM derivation, transformation, optimization, and low-level specialization down to the compiler framework, while generating trigger programs that achieve orders of magnitude better performance.

Experimental Environment. The experiments described in this section are conducted under two different configurations: *a)* **Local:** For moderate size experiments, we use a multiprocessor workstation environment with a 2.66GHz Intel Xeon with 2×6 cores, each with 2 hardware threads, 64GB of DDR3 RAM, and Mac OS X Lion 10.11.5. Dense BLAS operations are supported through the underlying Mac VecLib framework. *b)* **Distributed:** For large scale

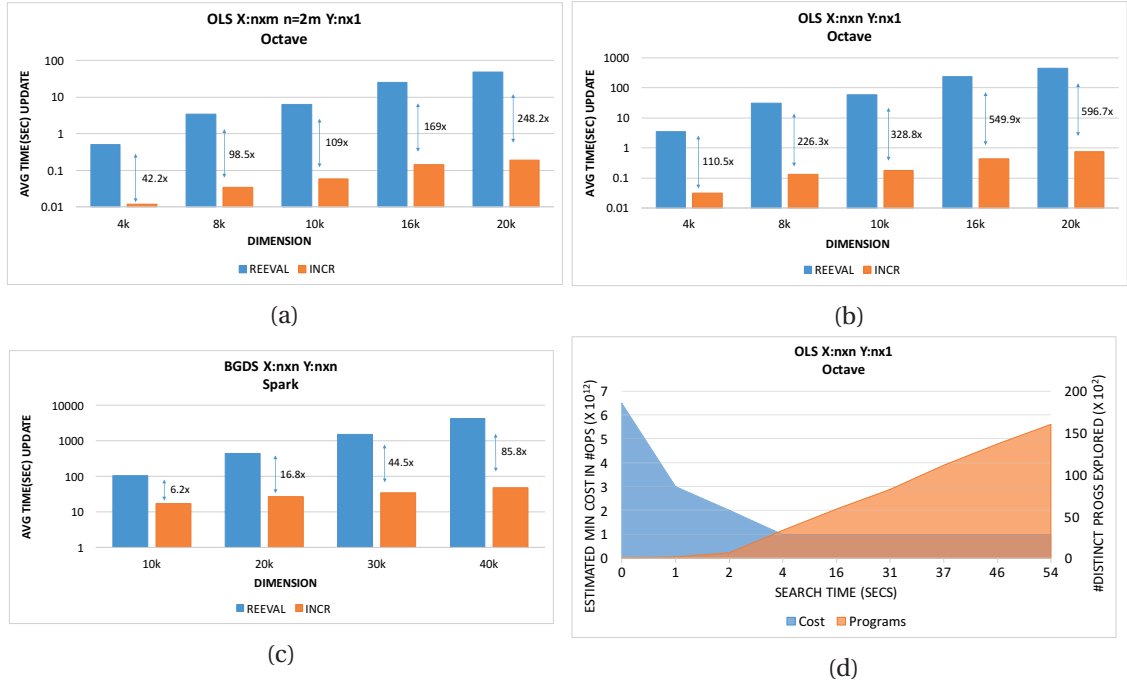


Figure 5.15 – Performance evaluation of Incremental Linear Regression.

experiments, we use a cluster of 100 server instances connected via a full-duplex 10GbE network and running Spark 1.6.1 and YARN 2.7.1. We compile Spark programs using Scala 2.10.4. Each instance is equipped with an Intel Xeon E5-2630L @ 2.40GHz server with 2×6 cores, each with 2 hardware threads, 15MB of cache, 128GB of DDR3 RAM, and Ubuntu 14.04.2 LTS. We rely on the ATLAS library to support multithreaded BLAS operations. For Spark, we have implemented a code generator for the subset of Lago required for these experiments. In this case, we implement a runtime using Spark RDDs that allow for mutable operations on block matrices that call efficient BLAS routines locally. All experiments on matrix RDDs have a predefined block distribution of 10×10 blocks. Efficient partitioning of matrices is orthogonal to the discussion of this chapter and can be handled by other systems like SystemML [78]. For example, Lago can generate SystemML matrix programs, i.e., compositions of matrix operations using the SystemML DSL, which is then handled and optimized by SystemML. For all IVM experiments, unless stated otherwise, We simulate a stream of rank-one updates to evaluate the performance of incremental view maintenance.

5.6.1 Incremental Linear Regression

In this set of experiments, we evaluate the performance of the common machine learning task of building a linear regression model given the independent and dependent variables

| | Iterations (#) | Compilation Time (s) | Rules (#) | Equivalence Rules (%) | Programs Revisited (%) |
|------|-------------------|-------------------------|--------------|--------------------------|---------------------------|
| OLS | - | 36 | 10869640 | 7% | 90.98% |
| BGDS | 4 | 0.567 | 4526 | 7% | 75.56% |
| BGDS | 16 | 0.944 | 28826 | 4% | 81.08% |
| BGDS | 128 | 15.139 | 1328854 | 0.8% | 87.76% |
| BGDS | 256 | 128.599 | 3841206 | 0.5% | 87.35% |

Table 5.2 – Report on compilation metrics.

X and Y respectively. In particular, we experiment on two programs: *a*) **OLS**: the Ordinary Least Squares method as illustrated in Section 5.4.1 to evaluate the statistical estimate β^* via the matrix expression $\beta^* := (X^T X)^{-1} X^T Y$, and *b*) **BGDS**: the Iterative Batch Gradient Descent method (BGDS) which, similar to OLS, evaluates the statistical estimate Θ that is computed via the recurrence relation $\Theta_{i+1} := \Theta_i - X^T (X\Theta_i - Y)$. Given ΔX changes, Lago derives the incremental version of each program and generates the corresponding trigger code. Furthermore, to demonstrate portability, we generate Octave code (**Local**) for OLS and Spark code (**distributed**) for BGDS. For comparison, we compare the re-evaluation of the original programs REEVAL against their corresponding derived trigger code INCR.

OLS Evaluation. We conduct a set of experiments to evaluate the statistical estimator β^* . The predictor matrix X has dimension $(n \times m)$ and the response matrix Y is of dimension $(n \times 1)$. Given a continuous stream of ΔX updates on X , Fig. 5.15a and Fig. 5.15b compare the average execution time per ΔX update of REEVAL with that of INCR for different values of n (x-axis). In particular, we experiment on two settings, in particular, when X is a tall skinny matrix with dimensions $n \times m$, where $n = 2m$ (Fig. 5.15a) and when X is a square matrix with dimensions $n \times n$ (Fig. 5.15b), i.e., $m = n$. The graphs illustrate how INCR outperforms REEVAL in computing the β^* estimate. Notice the asymptotic difference between the two, that is the performance gap between REEVAL and INCR widens as the matrix size increases, i.e., 42.2x — 248.2x and 110.5x — 596.7x respectively. The cost of REEVAL is dominated by costly $\mathcal{O}(n^3)$ matrix operations which include matrix inversion and multiplication, whereas, INCR avoids matrix inversions and evaluates cheaper $\mathcal{O}(n^2)$ matrix multiplications.

BGDS Evaluation. We also experiment on the batch gradient descent method to compute the estimate Θ for the linear regression problem. This method is usually used, instead of OLS, as a fast approximate or when the expression $X^T X$ is singular, i.e., non-invertible. For experimental purposes, we set X and Y with dimensions $(n \times n)$ and we fix the number of iterations to 16 assuming that the result converges to an appropriate solution after this number of steps. Fig. 5.15c demonstrates the average computation time per incremental update ΔX for each of REEVAL and INCR. The results demonstrate 6.2x-85.8x performance speedups as

the dimension size n increases. Distributed matrix multiplications require partitioning the matrices appropriately to evaluate the final result. This poses large communication overhead due to repartitioning. On the other hand, if one of the matrices undergoing multiplication is fairly small, e.g., vector, it is broadcasted to all partitions instead of repartitioning the bigger matrix. Given that the delta expressions in INCR are materialized in factored forms, their multiplications are much cheaper. Therefore, not only does INCR avoid costly matrix multiplication operations, but it also avoids expensive communication overheads.

Search Space. We also evaluate the explored search space using the traditional breadth-first-search (BFS) for both OLS and BGDS. OLS represents a small program and BGDS represents a big size program, i.e., defined by the number of iterations. We experiment on two different search configurations. For the OLS program with $n = 10000$ we run a complete BFS search on the whole derived delta program, whereas for BGDS, cost-based optimization complies with the phased approach described in Section 5.3.5 which confines the search on each statement independently from the other statements. Fig. 5.15d illustrates two dimensions against elapsed search time. First, the number of distinct programs explored and secondly, the minimum inferred cost of the explored programs. The cost here depicts the sum of costs for both the original and the trigger program. Notice how the minimum cost decays fast during the early stages of the search as more programs are being explored.

The search begins with an initial program as illustrated in Fig. 5.13a. The original program requires 2 matrix multiplications, 1 matrix-vector multiplication, 1 matrix inverse, and 2 matrix transposes. That is a sum of $3n^3$ and $3n^2$ operations. Moreover, the initial trigger program, which is achieved by naïvely replacing each X with $X + \Delta X$, requires $3n^3$ and $6n^2$ operations. All in all, this requires $6n^3 + 9n^2$ operations which when substituted with $n = 10000$ gives around 6 billion operations as depicted in the figure at time 0. At time 1, the search algorithm is able to transform all expensive operations $\mathcal{O}(n)^3$ in the trigger program to cheaper $\mathcal{O}(n)^2$ ones as described in Fig. 5.13o. Then the total cost is reduced to that of the original program (pre-computations) which accounts for 3 billion operations as depicted in Fig. 5.15d. Interestingly, the search algorithm finds a simple equivalent program (to that of Fig. 5.13a) as follows $\beta := X^{-1}Y$. Although the program is numerically unstable in comparison to computing the pseudoinverse $(X^T X)^{-1}$, it is analytically equivalent to the original program and it is much cheaper to evaluate as it only requires computing one matrix inverse (n^3). The program is found at time 4 secs in Fig. 5.15d.

The search reaches a point where it introduces negligible savings. The search algorithm finds the minimal cost at second 36, after it has explored a search space created from applying 10,105,018 simplification rules and 764,622 equivalence rules. To avoid re-visiting the same programs within the search space, we maintain a cache that saves the hash-codes of the canonical representation of visited programs, i.e. canonical representation of the IR tree.

This saves a lot from doing redundant work. For instance after 36 seconds the cache reports 90.983% hits and 9.016% misses. This suggests that a large number of the generated candidate programs have been explored before, which also means that many of the different orderings of the transformation rules yield the same programs.

Table 5.2 illustrates various compilation metrics on the derivation and optimization of OLS and BGDS. Although OLS is a small program, the search uses a large number of rules to explore the search space. This is because search is applied on all the statements of the programs at once. On the other hand, BGDS has a more confined search space as it optimizes each statement independently. This phased approach works well with large size programs.

5.6.2 Graph Analytics

Many graph computations can be formulated as matrix operations. In this section, we experiment on the all pairs k -hop reachability/shortest path problem for the undirected graph G . The program, in Fig. 5.7, is the same as the running example used through out this chapter. As mentioned earlier, different semiring configurations for this program result in different programs. In particular, a Boolean semiring $\langle \{0, 1\}, \vee, \wedge \rangle$ defines the reachability problem whereas the tropical semiring $\langle \mathbb{R}, \min, + \rangle$ defines the shortest path problem.

Meta-Information Specialization

As explained earlier in section 5.3.4, G is an undirected binary graph. This meta-information can be encoded into the input data and Lago propagates this information and tries to infer properties for the subsequent and intermediate statements. Meta-information leverages specialization opportunities at the code generation phase. For instance, in the graph reachability program example, the following specialization opportunities are possible: *a) Bit vectors:* The domain values are either zeros or ones only. Accordingly, rather than representing the adjacency matrix entries using the more generic single or double-precision types, one can utilize compressed bit vectors to pack every eight cells into a single byte. As we will demonstrate later, this compacts storage, allows for large matrix constructions, and avoids expensive communication costs for data shuffling in a distributed setting. *b) Boolean Algebra:* The semiring operations, i.e., Boolean conjunction and disjunction, enable Boolean algebra optimizations and specialization opportunities. For instance, the dot product of two vectors can be translated as computing the bitwise-AND of the two bit vectors followed by evaluating if the result is bigger than zero. This leverages vectorized operations. Furthermore, one can benefit from short-circuiting rather than passing over the entire bit vectors to compute the dot product. Matrix multiplication $G \times G$ can be specialized along the same lines. Alternatively, in a general purpose environment, i.e. R or Matlab, this expression is evaluated using the

following expression $(G \times G) > 0$. That is, it computes the matrix multiplication numerically first and then a logical indexing is applied over the result matrix to bring it back to the binary domain. c) **Symmetry**: Matrix symmetry enables many specializations ranging from compact representation, e.g., lower triangular, to calling specialized matrix operations that exploit symmetry. In this evaluation, we focus on a specific specialization that leverages the matrix layout. Since the bit vector matrices can be represented as rows or columns of bit vectors, there are two layout configurations, i.e., row-major layout and column-major layout. The operations on the matrices define the ideal layout representation of these matrices. For example, consider $G \times G$, ideally matrix G should have both row-major and column-major layouts to support direct use of the bitwise operations; otherwise transformations to the layout should be applied which incurs cost. Fig. A.7 depicts the inference rules for bit vector layouts. However, if G is symmetric then $G^T = G$, which means that matrix G represents both logical layouts, independently from its underlying physical representation. This eliminates the need for transforming G 's layouts and therefore its associated costs.

Specialization Evaluation. Graph analysis in this domain relies on matrix algebra operations and most notably on matrix multiplication which is commonly used in graph clustering, betweenness centrality, graph contraction, subgraph extraction, cycle detection, and quantum chemistry [38, 165, 66]. To that end, we focus our attention on the microbenchmark of evaluating the performance of specialized sparse matrix multiplications. We experiment on two different settings:

Local. We compare between four different specialized implementations in Scala: a) `SymBit` represents the implementation of all the previous specializations. b) `Bit` is similar to `SymBit`, but excludes the symmetry specialization. c) `CSC` represents the implementation using the conventional Compressed Column Storage format [150]. This format is mainly used for sparse matrices and it maintains matrix values along with their indexes in a compact form. d) `Dense` represents the multithreaded general purpose implementation that calls native dense BLAS routines for double precision operations. Notice that the following evaluation results are for a single thread except for `Dense` that leverages its native multithreading capabilities.

For the first set of experiments, we evaluate the potential of the aforementioned specializations. To that end, we focus on the micro-benchmarks of a single matrix-matrix multiplication $G \times G$. The input binary matrix is randomly generated with density 50%. We set this density configuration because the reachability program results in denser (intermediate) results after a few iterations (hops). Fig. 5.16a reports the average execution time for each implementation with varying dimension size n . First, let us compare the general purpose implementations `CSC` and `Dense`. Notice how `CSC` performs poorly in comparison to `Dense` as n grows and how it begins to fail after 20k. This is because of the high matrix density, which makes `CSC` inefficient for storage, i.e., saving index information, and for computation, i.e., no cache

5.6. Evaluation

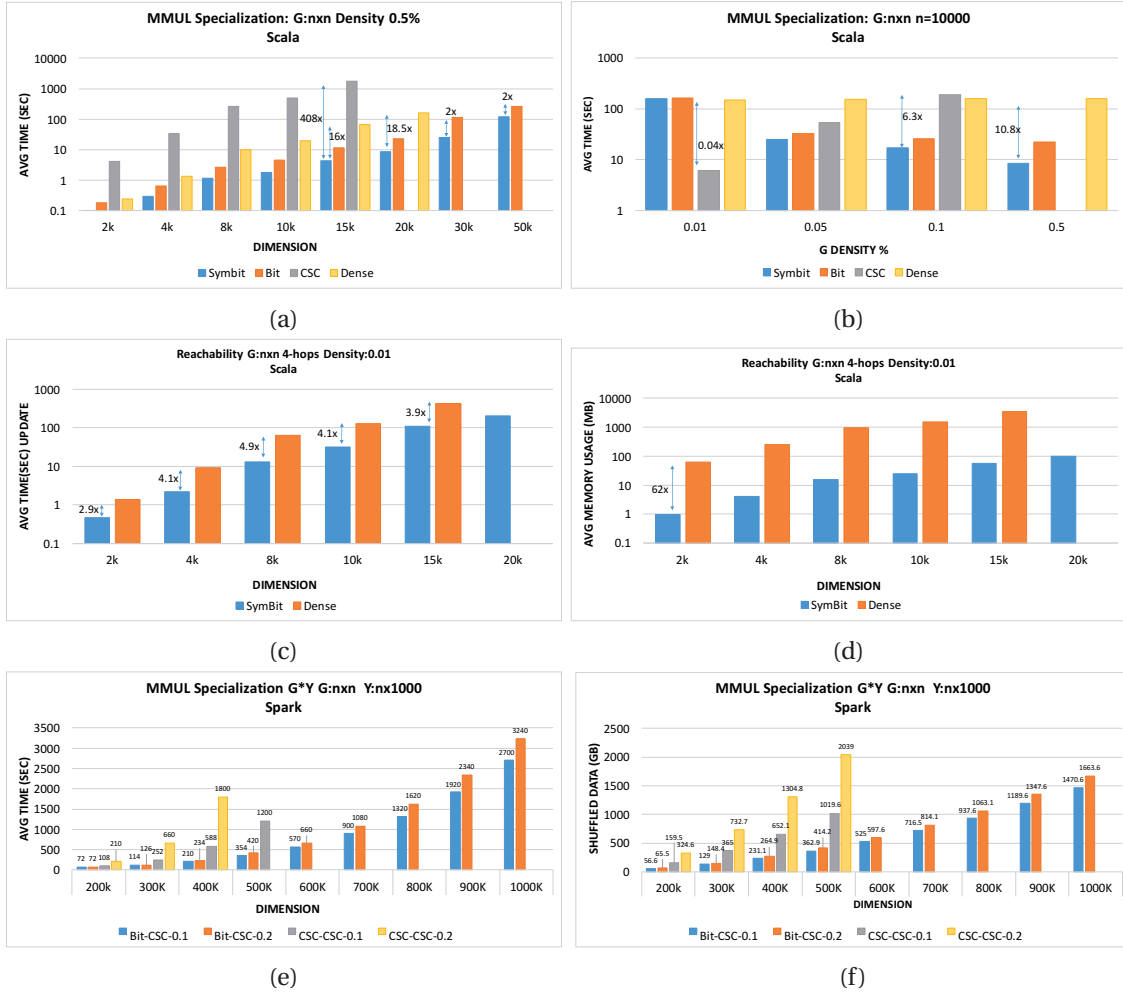


Figure 5.16 – Performance Evaluation of Meta-Information specialization opportunities.

locality. Moreover, CSC computes on one core whereas Dense leverages all the available cores. On the other hand, the bit vector implementations Bit and SymBit exhibit scalable performance. They can scale to larger sizes n while maintaining a very compact storage representation up to $n = 100k$. Moreover, they benefit from short-circuiting given the density of the matrices. This saves from passing over all the entries within the whole matrix and achieves more than two order of magnitudes better performance than CSC and one order of magnitude better than Dense with one core only. Moreover, SymBit exploits the symmetric property and has 2x better performance than Bit.

To explore the effect of Graph density on the previous implementations, we fix the dimensions size $n = 10k$ and vary the density parameter. Fig. 5.16b illustrates the results. At the density level 0.01, CSC beats all the others due to the sparsity of the input which makes this format

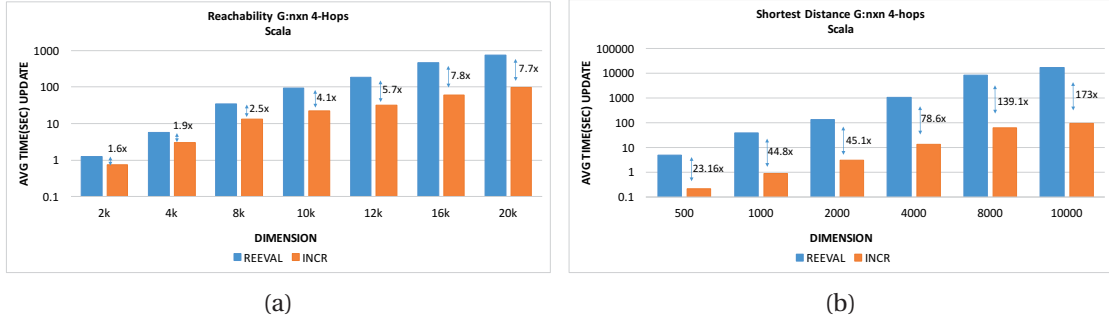


Figure 5.17 – Performance Evaluation of Incremental Graph programs using Symbit for Reachability and Dense for Shortest Distance.

and implementation the most suitable. SymBit and Bit cannot leverage short-circuiting at this stage due to sparsity. However, as the density increases SymBit and Bit outperform the others. Notice how the performance of Dense does not change, as it is agnostic to the underlying structure of the input matrices.

Putting it all together, we experiment on the overall reachability program on randomly generated scale-free graphs G with density 0.01. Fig. 5.16c and Fig. 5.16d present the execution time and space utilization respectively. SymBit outperforms Dense by up to 3x-5x in performance and up to 62x in space savings. The reduction in space is consistent with the fact that bit vectors allow compacting 64 item into 8 bytes rather than a double value that represents a single item in 8 bytes.

Distributed. In this experiment, we evaluate the large scale matrix multiplication $G \times Y$ under the numerical semiring, where G is a binary graph and Y is a matrix with arbitrary values. This operation is common in graph algorithms such as vertex clustering [79]. We compare between two implementations in Spark: *a*) Bit-CSC and *b*) CSC-CSC. Bit-CSC represents the first matrix in bit vector format and the other matrix in CSC. We experiment on graphs with two density settings 0.1 and 0.2 with variable dimension size n . Fig. 5.16e and Fig. 5.16f demonstrate how the specialized code Bit-CSC outperforms CSC-CSC as n grows. The performance gains are pronounced in the communication savings of shuffling compressed bit vectors rather than larger unnecessary general purpose datastructures. Since communication dominates cost in a distributed environment, these savings result in better resource utilization and performance. Notice how Bit-CSC can scale to large graphs. In summary, Lago leverages useful meta-information that opens up opportunities for optimization at the code generation phase. As we have demonstrated, these optimizations can range from datastructure to computation specializations.

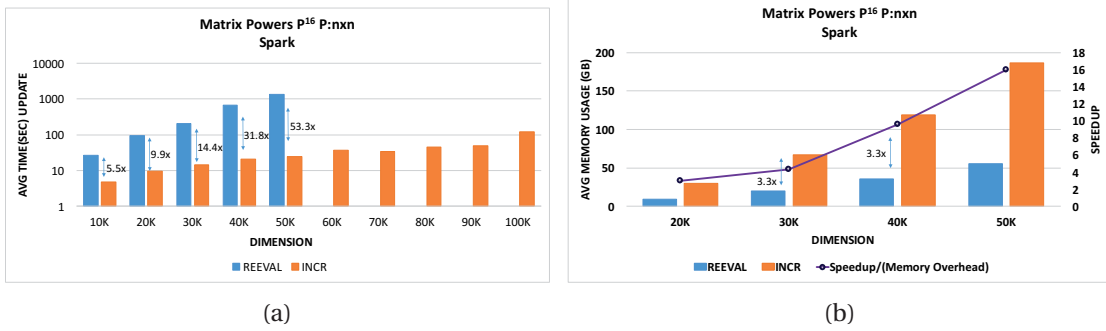


Figure 5.18 – Scalability and additional storage results.

Incremental Graph Analytics

By combining semiring configurations with IVM capability as described in 5.6.1, one can directly derive incremental graph analytics. Notice that the delta rules operate at the abstract level of matrix algebra operations. This permits reasoning about the derivation of incremental programs at a high level without delving into the details of the underlying operations used within the matrix operators i.e., semiring.

Evaluation. Fig. 5.17a compares the update performance of the previously described 4-hops reachability problem against its incremental version as generated by Lago. Similarly, Fig. 5.17b compares the update performance for the 4-hops shortest path problem. Lago is able to derive the incremental program of the matrix program independent from the underlying semiring semantics. Again, we can observe performance gains for IVM in comparison to re-evaluation, i.e., 1.6x — 7.7x in the case of reachability and 23.16x — 173x in the case of shortest paths. Notice how the performance gains in the boolean semiring are not as big as the other experiments. This is because of short-circuiting which introduces large performance gains that are comparable to the gains of IVM. Also notice how the shortest path program is much more slower than that of reachability, although they only differ in the semiring definition. The reason is that the Tropical semiring requires evaluating the **min** operator $\mathcal{O}(n^3)$ times. There is no specialized machine instruction for this operator as opposed to addition/multiplication in the numerical semiring and vectorwise and/or in the boolean semiring. Therefore, the **min** operator is expanded to many other machine instructions which is even much more costly in a loop, i.e. matrix multiplication.

5.6.3 Scalability Evaluation

In this section we evaluate several dimensions of IVM in comparison to re-evaluation, in particular, scalability, memory consumption of materialized view and performance of batch updates. For these experiments we evaluate on the matrix powers problem on dense matrices,

| Zipf factor | 5.0 | 4.0 | 3.0 | 2.0 | 1.0 | 0.0 |
|--------------|------|------|------|-------|-------|--------|
| Octave (10K) | 6.3 | 6.8 | 7.5 | 10.9 | 68.4 | 236.5 |
| Spark (30K) | 28.1 | 41.5 | 67.3 | 186.1 | 508.9 | 1678.8 |

Table 5.3 – The average Octave and Spark view refresh times in seconds for INCR of P^{16} and a batch of 1,000 updates. The row update frequency is drawn from a Zipf distribution.

in particular we compute the matrix power P^{16} on Spark. Fig. 5.18a shows how incremental evaluation outperforms evaluation as previous results. Moreover, INCR can scale to larger dimensions, i.e. $n = 100k$, whereas REEVAL cannot go beyond 50k, as the sizes of shuffled data for matrix multiplication increase resulting in large communication overheads and unmaintainable states at each machine.

On the other hand, IVM requires additional storage for maintaining materialized views of intermediate results. Fig. 5.18b demonstrates the average memory usage of INCR in comparison to REEVAL. INCR consistently uses 3.3x more storage no matter the dimension n , that is because the program maintains 4 intermediate results, in particular P^2 , P^4 , P^8 , and the result P^{16} . To compare the performance speedup gains in comparison to the costs of extra storage, the figure also demonstrates the ratio between (speedup gain)/(storage-cost). The results show how the gains ratio keep on increasing with the dimensions size, i.e., 3x — 16x. This is consistent with the asymptotic increase in the computational gain versus the constant increases in storage costs.

In the final set of experiments we explore the efficiency of IVM under batches of updates. We simulate a use case where some regions (rows) of the matrix P are updated more frequently than others. The frequency of updates is set by a Zipf distribution that is controlled by the Zipf exponent factor. When the factor value increases, it simulates a more skewed distribution, on the other hand, if it decreases it converges more towards a uniform distribution of changes. Table 5.3 reports on the performance results of IVM of P^{16} under a batch of 1000 tuples. As the Zipf factor tends to a uniform distribution, the overall rank of the updates increases and thus IVM loses its benefit in comparison to re-evaluation. To put the results in context, the cost of re-evaluation is 99.1 seconds and 203.4 for Octave and Spark respectively.

6 Conclusion

This research is motivated by the need for supporting a wider class of advanced analytics on top of real-time data streams. In this thesis, we present foundations and techniques that enable online query processing to efficiently support relational joins with arbitrary join-predicates beyond traditional equi-joins; and to support other data models that target machine learning and graph computations. This thesis is based on two main research directions:

Chapter 4 provides a novel adaptive solution to computing joins with general predicates in an online setting. Unlike previous offline approaches, the adaptive operator presented does not require any prior knowledge about the input data. This is essential when statistics about input data are not known in advance or are difficult to estimate. The operator is highly scalable and continuously processes input streams even during adaptation. Theoretical analysis proves that our algorithm maintains a close-to-optimal state, under an experimentally validated performance measure that captures resource utilization. Furthermore, cost of adaptation is provably minimum. Experiments validate the theoretical guarantees and show that the operator outperforms static approaches; is highly adaptive; and is resilient to data skew. It is also very efficient in resource consumption and maintains high throughput and low tuple latency. Evaluation suggests that there is room for optimization for a special class of joins like equi and band joins. In such low-selectivity joins, the join matrix contains large regions where the join condition never holds. These regions need not be assigned joiners. This motivates designing a content-sensitive theta-join operator. Such an operator shares many common features with our operator, but its design poses additional challenges (Section 4.4.4).

Chapter 5 presents Lago, a novel framework that supports the Incremental View Maintenance of matrix algebra workloads. Linear algebra represents a concrete substrate for advanced analytical tasks including, machine learning, scientific computation, and graph algorithms. We show how previous works on IVM are not applicable to matrix algebra workloads, as a single entry change to an input-matrix results in changes all over the intermediate views, rendering

IVM useless in comparison to re-evaluation. Lago automatically derives and optimizes incremental trigger programs of analytical computations, while freeing the user from erroneous manual derivations, low-level implementation details, and performance tuning. We present a novel technique that captures Δ changes as low-rank matrices which are representable in a compressed factored form that enables cheaper computations. Lago automatically propagates the factored representation across program statements to derive an efficient trigger program. we present and evaluate the IVM of several practical use case examples including computing linear regression models, gradient descent, and all-pairs graph reachability or shortest path computations. The evaluation results demonstrate orders of magnitude (10x-100x) better performance in favor of derived trigger programs in comparison to simple re-evaluation. Future work includes extending the language to support other matrix algebra operations, e.g., element-wise operations and matrix factorizations.

A Appendix

A.1 Analysis under Window Semantics

Algorithm 5 Migration Decision Algorithm (with deletions).

Input: $|R|, |S|, |\Delta R|, |\Delta S|$

```
1: function MIGRATIONDECISION( $|R|, |S|, |\Delta R|, |\Delta S|$ )
2:   if  $|\Delta R| \geq |R|/2$  or  $|\Delta S| \geq |S|/2$  then
3:     Choose mapping  $(n, m)$  minimizing  $\frac{|R|+|\Delta R|}{n} + \frac{|S|+|\Delta S|}{m}$ 
4:     Decide a migration to  $(n, m)$ 
5:      $|R| \leftarrow |R| + |\Delta R|; |S| \leftarrow |S| + |\Delta S|$ 
6:      $|\Delta R| \leftarrow 0; |\Delta S| \leftarrow 0$ 
7:   end function
```

Theorem A.1.1. Assume that the number of joiners J is a power of two, the sizes for $|R|$ and $|S|$ are no more than a factor of J apart, and that tuples from R and S have the same size. An adaptive scheme that applies Alg. 5 ensures the following characteristics:

1. The ILF is at most 1.4 times that of the optimal mapping at any point in time. $ILF \leq 1.4 \cdot ILF^*$, where ILF^* is the input-load factor under the optimal mapping at any point in time. Thus, the algorithm is 1.4-competitive.
2. The total communication overhead of migration is amortized, i.e., the cost of routing a new input tuple, including its migration overhead, is $O(1)$.

Alg. 5 decides migration once $|\Delta R| = |R|/2$ or $|\Delta S| = |S|/2$. Notice that $|\Delta R|$ and $|\Delta S|$ include insertions and deletions. Therefore, lemma 4.4.3 implies that while the system is operating with the mapping (n, m) , the optimum is one of (n, m) , $(n/2, 2m)$, and $(2n, m/2)$. This implies the following.

Appendix A. Appendix

Lemma A.1.1. *If $|\Delta R| \leq |R|/2$ and $|\Delta S| \leq |S|/2$ and (n, m) is the optimal mapping for $(|R|, |S|)$ tuples, then under Alg. 5, the input-load factor ILF never exceeds $1.4 \cdot ILF^*$. In other words, the algorithm is 1.4-competitive.*

Proof. By lemma 4.4.3, the optimal mapping is either (n, m) , $(n/2, 2m)$ or $(2n, m/2)$. If the optimal mapping is (n, m) then $ILF = ILF^*$. Otherwise, the ratio can be bounded as follows. Without loss of generality, assume that the optimum is $(n/2, 2m)$ then

$$\frac{ILF}{ILF^*} \leq \frac{(|R| + |\Delta R|)/n + (|S| + |\Delta S|)/m}{(|R| + |\Delta R|)/(n/2) + (|S| + |\Delta S|)/(2m)}$$

where the constraints $|\Delta R|/n \leq |R|/n$, $|\Delta S|/m \leq |S|/m$ and those in lemma 4.4.2 must hold. Final cardinalities are non-negative. Consider the ratio as a function of the variables $|R|/n$, $|S|/m$, $|\Delta R|/n$ and $|\Delta S|/m$. The maximum value of the ratio of linear functions in a simplex (defined by the linear constraints) is attained at a simplex vertex. By exhaustion, the maximum occurs when $|\Delta R| = -|R|/2$, $|\Delta S| = |S|/2$ and $|S|/m = 2|R|/n$. Substituting gives 1.4. \square

Lemma A.1.2. *The cost of routing tuples and data migration is linear. The amortized cost of an input tuple is $O(1)$.*

Proof. Since all joiners are symmetrical and operate simultaneously in parallel, it suffices to analyze cost at one joiner. Therefore, after receiving $|\Delta R|$ and $|\Delta S|$ tuples, the operator spends at least $\max(|\Delta R|/n, |\Delta S|/m)$ units of time processing these tuples at the appropriate joiners. By assigning a sufficient amortized cost per time unit, the received tuples pay for the later migration.

By lemma 4.4.3, the optimal mapping is (n, m) , $(n/2, 2m)$ or $(2n, m/2)$. If the optimal mapping is (n, m) , then there is no migration. Without loss of generality, assume that $|\Delta S| \geq |\Delta R|$ and that the optimal mapping is $(n/2, 2m)$. Between migrations, $\max(|\Delta R|/n, |\Delta S|/m)$ time units elapse, each is charged 11 units. One unit is used to pay for routing and 10 are reserved for the next migration. The cost of migration by lemma 4.4.5 is $2(|R| + |\Delta R|)/n$. The amortized cost reserved for migration is $6 \max(|\Delta R|/n, |\Delta S|/m)$. Since a migration was triggered, either $|\Delta R| = |R|$ or $|\Delta S| = |S|$. In either case, it should hold that the reserved cost is at least the migration cost, that is,

$$10 \max(|\Delta R|/n, |\Delta S|/m) \geq 2(|R| + |\Delta R|)/n.$$

If $|\Delta R| = |R|/2$, then by substituting, the left hand side is $10 \max(|\Delta R|/n, |\Delta S|/m) \geq 10|R|/n$ and the right hand side is $2(|R| + |\Delta R|)/n = 4|R|/n$. Therefore, the inequality holds. If $|\Delta S| =$

$|S|/2$, then the left hand side is

$$10 \max(|\Delta R|/n, |\Delta S|/m) \geq |\Delta R|/n + 4|S|/m.$$

Therefore, the left hand side is not smaller than the right hand side, since $2|S|/m \geq |R|/n$ (by lemma 4.4.2). Thus, the inequality holds in both cases. The cases, when $|\Delta R| \geq |\Delta S|$ or when the optimal is $(2n, m/2)$, are symmetric. \square

Lemma A.1.3. *Lemmas A.1.1 and A.1.2 directly imply Theorem A.1.1.*

A.2 LAGO Rules

$$\begin{array}{c}
 \frac{m : S}{m^\top \rightarrow m} \quad \frac{}{(m^\top)^\top \rightarrow m} \quad \frac{m : \mathcal{D}_{(r,c)}}{\mathbf{cols}(m) \rightarrow c} \quad \frac{m : \mathcal{D}_{(r,c)}}{\mathbf{rows}(m) \rightarrow r} \quad \frac{}{\mathbf{iterate}[0](m)(x \Rightarrow f(x)) \rightarrow m} \\
 \text{ITERATE-UNROLL} \\
 \frac{n > 0}{\mathbf{iterate}[n](m_0)(x \Rightarrow f(x)) \rightarrow \mathbf{let } x_0 = f(m_0) \mathbf{ in } \mathbf{iterate}[n-1](x_0)(x \Rightarrow f(x))} \\
 \\
 \frac{}{\mathbf{let } x_1 = x_2 \mathbf{ in } m_1 \rightarrow m_1 [x_1 := x_2]} \quad \frac{\text{occurrences}(m_2, x) = 1}{\mathbf{let } x = m_1 \mathbf{ in } m_2 \rightarrow m_2 [x_1 := m_1]} \quad \frac{\text{occurrences}(m_2, x) = 0}{\mathbf{let } x = m_1 \mathbf{ in } m_2 \rightarrow m_2} \\
 \\
 \frac{m : \mathcal{D}_{(r,k)}}{m \cdot \mathbf{zeros}[k, c] \rightarrow \mathbf{zeros}[r, c]} \quad \frac{}{m + \mathbf{zeros}[r, c] \rightarrow m} \quad \frac{m : \mathcal{D}_{(r,c)}}{m \cdot \mathbf{id}[c] \rightarrow m} \\
 \frac{m : \mathcal{D}_{(k,c)}}{\mathbf{zeros}[r, k] \cdot m \rightarrow \mathbf{zeros}[r, c]} \quad \frac{}{\mathbf{zeros}[r, c] + m \rightarrow m} \quad \frac{m : \mathcal{D}_{(r,c)}}{\mathbf{id}[r] \cdot m \rightarrow m}
 \end{array}$$

Figure A.1 – Simplification rules

| | | |
|--|---|--|
| $\frac{}{\mathbf{m}_1 + \mathbf{m}_2 \leftrightarrow \mathbf{m}_2 + \mathbf{m}_1} \text{ADD-COMM}$ | $\frac{}{(\mathbf{m}_1 + \mathbf{m}_2) + \mathbf{m}_3 \leftrightarrow \mathbf{m}_1 + (\mathbf{m}_2 + \mathbf{m}_3)} \text{ADD-ASSOC}$ | $\frac{}{(\mathbf{m}_1 \cdot \mathbf{m}_2) \cdot \mathbf{m}_3 \leftrightarrow \mathbf{m}_1 \cdot (\mathbf{m}_2 \cdot \mathbf{m}_3)} \text{MULT-ASSOC}$ |
| $\frac{}{\mathbf{m}_1 \cdot (\mathbf{m}_2 + \mathbf{m}_3) \leftrightarrow \mathbf{m}_1 \cdot \mathbf{m}_2 + \mathbf{m}_1 \cdot \mathbf{m}_3} \text{DISTRIB}$ | | |
| $\frac{}{(\mathbf{m}_1 + \mathbf{m}_2)^\top \leftrightarrow \mathbf{m}_1^\top + \mathbf{m}_2^\top} \quad \frac{}{(\mathbf{m}_1 \cdot \mathbf{m}_2)^\top \leftrightarrow \mathbf{m}_2^\top \cdot \mathbf{m}_1^\top} \quad \frac{}{(\mathbf{m}_1 \Rightarrow \mathbf{m}_2)^\top \leftrightarrow \mathbf{m}_1^\top \Downarrow \mathbf{m}_2^\top}$ | | |
| $\frac{}{\mathbf{let } x = \mathbf{m}_1 \mathbf{ in } \mathbf{m}_2 \leftrightarrow \mathbf{m}_2 [x_1 := \mathbf{m}_1]} \text{LET-INLINE}$ | | |
| $\frac{}{\mathbf{m}_1 : \mathcal{D}_{(n,k)} \quad \mathbf{m}_2 : \mathcal{D}_{(k,m)} \quad \mathbf{m}_3 : \mathcal{D}_{(n,p)} \quad \mathbf{m}_4 : \mathcal{D}_{(p,m)}} \text{FACTORIZATION}$ | | |
| $\frac{}{\mathbf{m}_1 \cdot \mathbf{m}_2 + \mathbf{m}_3 \cdot \mathbf{m}_4 \leftrightarrow (\mathbf{m}_1 \Rightarrow \mathbf{m}_3) \cdot (\mathbf{m}_2 \Downarrow \mathbf{m}_4)} \text{WOODBURY FORMULA}$ | | |
| $\frac{}{\mathbf{m}_1 : \mathcal{D}_{(n,n)} \quad \mathbf{m}_2 : \mathcal{D}_{(n,k)} \quad \mathbf{m}_3 : \mathcal{D}_{(k,n)}} \text{WOODSBURY FORMULA}$ | | |
| $\frac{}{(\mathbf{m}_1 + \mathbf{m}_2 \cdot \mathbf{m}_3)^{-1} \leftrightarrow \mathbf{m}_1^{-1} - \mathbf{m}_1^{-1} \cdot \mathbf{m}_2 \cdot (\mathbf{id}[k] + \mathbf{m}_3 \cdot \mathbf{m}_1^{-1} \cdot \mathbf{m}_2)^{-1} \cdot \mathbf{m}_3 \cdot \mathbf{m}_1^{-1}}$ | | |

Figure A.2 – Equivalence rules

$$\frac{\mathbf{m}_1 : \mathcal{S} \quad \mathbf{m}_2 : \mathcal{S} \quad \forall a, b. a \circ b = b \circ a}{\mathbf{m}_1 \circ \mathbf{m}_2 : \mathcal{S}} \quad \frac{\mathbf{m} : \mathcal{D}_{(1,1)}}{\mathbf{m} : \mathcal{S}} \quad \frac{\mathbf{m} : \mathcal{S}}{\mathbf{m}^\top : \mathcal{S}} \quad \frac{\mathbf{m} : \mathcal{S} \quad k \geq 0}{\mathbf{m}^k : \mathcal{S}}$$

 Figure A.3 – Inferring **symmetry** of matrices.

$$\frac{\mathbf{m}_1 : \mathcal{D} \quad \mathbf{m}_2 : \mathcal{D}}{\mathbf{m}_1 \circ \mathbf{m}_2 : \mathcal{D}} \quad \frac{\mathbf{m}_1 : \mathcal{D} \quad \mathbf{m}_2 : \mathcal{D}}{\mathbf{m}_1 \times \mathbf{m}_2 : \mathcal{D}}$$

 Figure A.4 – Inferring **Sparse Structures** of matrices.

$$\frac{\mathbf{m} : \mathcal{D}_{(r,c)}}{\mathbf{m} : \mathcal{R} \leq \min(r, c)} \quad \frac{\mathbf{m}_1 : \mathcal{R}_1 \quad \mathbf{m}_2 : \mathcal{R}_2}{\mathbf{m}_1 \cdot \mathbf{m}_2 : \mathcal{R} \leq \min(\mathcal{R}_1, \mathcal{R}_2)} \quad \frac{\mathbf{m}_1 : \mathcal{R}_1 \quad \mathbf{m}_2 : \mathcal{R}_2}{\mathbf{m}_1 + \mathbf{m}_2 : \mathcal{R} \leq \mathcal{R}_1 + \mathcal{R}_2} \quad \frac{\mathbf{m} : \mathcal{R}}{\mathbf{m}^\top : \mathcal{R}} \quad \frac{\mathbf{m} : \mathcal{R}}{\mathbf{m}^\top \cdot \mathbf{m} : \mathcal{R}}$$

 Figure A.5 – Inferring **Ranks** of matrices.

$$\frac{\mathbf{m}_1 : \mathcal{L} \quad \mathbf{m}_2 : \mathcal{L}}{\mathbf{m}_1 + \mathbf{m}_2 : \mathcal{L}} \quad \frac{\mathbf{m}_1 : \mathcal{L} \quad \mathbf{m}_2 : \mathcal{L}}{\mathbf{m}_1 \times \mathbf{m}_2 : \mathcal{L}} \quad \frac{\mathbf{m} : \mathcal{L}}{\mathbf{m}^{-1} : \mathcal{L}} \quad \frac{\mathbf{m} : \mathcal{L}}{\mathbf{m}^\top : \mathcal{U}} \quad \frac{\mathbf{m}_1 : \mathcal{U} \quad \mathbf{m}_2 : \mathcal{U}}{\mathbf{m}_1 + \mathbf{m}_2 : \mathcal{U}} \quad \frac{\mathbf{m}_1 : \mathcal{U} \quad \mathbf{m}_2 : \mathcal{U}}{\mathbf{m}_1 \times \mathbf{m}_2 : \mathcal{U}} \quad \frac{\mathbf{m} : \mathcal{U}}{\mathbf{m}^{-1} : \mathcal{U}} \quad \frac{\mathbf{m} : \mathcal{U}}{\mathbf{m}^\top : \mathcal{L}}$$

 Figure A.6 – Inferring **structure** of Triangular matrices (\mathcal{U} : Upper triangle, \mathcal{L} : Lower triangle)

$$\frac{\mathbf{m}_1 : \mathcal{R} \quad \mathbf{m}_2 : \mathcal{R}}{\mathbf{m}_1 + \mathbf{m}_2 : \mathcal{R}} \quad \frac{\mathbf{m}_1 : \mathcal{C} \quad \mathbf{m}_2 : \mathcal{C}}{\mathbf{m}_1 + \mathbf{m}_2 : \mathcal{C}} \quad \frac{}{\mathbf{m}_1 \cdot \mathbf{m}_2 : \mathcal{R}} \quad \frac{\mathbf{m} : \mathcal{R}}{\mathbf{m}^\top : \mathcal{C}} \quad \frac{\mathbf{m} : \mathcal{C}}{\mathbf{m}^\top : \mathcal{R}} \quad \frac{}{\mathbf{m}_1 \Rightarrow \mathbf{m}_2 : \mathcal{C}} \quad \frac{}{\mathbf{m}_1 \Downarrow \mathbf{m}_2 : \mathcal{R}}$$

 Figure A.7 – Inferring **layout** of bit vector matrices (\mathcal{R} : Row layout, \mathcal{C} : Column layout)

Bibliography

- [1] Amazon Architecture. <http://highscalability.com/amazon-architecture>.
- [2] Computing at CERN. <https://home.cern/about/computing>.
- [3] The design of telegraph: Adaptive dataflow for streams. <http://db.cs.berkeley.edu/jmh/tmp/teleover-draft.pdf/>.
- [4] Scalding: A scala api for cascading. <https://github.com/twitter/scalding>.
- [5] The Apache Hadoop project. <http://hadoop.apache.org>.
- [6] The TPC-H benchmark. <http://www.tpc.org/tpch/>.
- [7] 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.
- [8] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), 2003.
- [9] U. Acar, G. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *TOPLAS*, 32(1), 2009.
- [10] M. R. Ackermann, M. Mörtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler. StreamKM++: A clustering algorithm for data streams. *J. Exp. Algorithmics*, 17, 2012.
- [11] E. Afrati and J. Ullman. Optimizing joins in a MapReduce environment. In *EDBT*, 2010.
- [12] C. C. Aggarwal and C. K. Reddy. *Data Clustering: Algorithms and Applications*. 2013.
- [13] F. Akgul. *ZeroMQ*. Packt Publishing, 2013.
- [14] 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 *VLDB*, pages 734–746, 2013.

Bibliography

- [15] 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. *VLDB*, 8(12), 2015.
- [16] 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.
- [17] E. Almeida, C. Ferreira, and J. a. Gama. Learning model rules from high-speed data streams. In *Proceedings of the 3rd International Conference on Ubiquitous Data Mining - Volume 1088*, UDM, 2013.
- [18] Amazon. Amazon Kinesis. <http://aws.amazon.com/kinesis/>.
- [19] Amazon. AWS Case Study: Supercell. <http://aws.amazon.com/solutions/case-studies/supercell/>.
- [20] Apache Flink: Scalable batch and stream data processing. <https://flink.apache.org/>.
- [21] 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.
- [22] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, 2005.
- [23] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*. Springer, 2002.
- [24] P. Baumann, A. Dehmelt, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. In *SIGMOD*, 1998.
- [25] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, 2014.
- [26] J. Berger. *Statistical decision theory and Bayesian analysis*. Springer, 1985.
- [27] J. Beringer and E. Hüllermeier. Online clustering of parallel data streams. *Data Knowl. Eng.*, 58(2), 2006.
- [28] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM infosphere streams for scalable, real-time, intelligent transportation services. In *SIGMOD*, 2010.

-
- [29] A. Bifet and R. Gavaldà. Adaptive learning from evolving data streams. In *Proceedings of the 8th International Symposium on Intelligent Data Analysis: Advances in Intelligent Data Analysis VIII*. Springer, 2009.
 - [30] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.
 - [31] 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.
 - [32] C. Bockermann. A Survey of the Stream Processing Landscape. Technical report, TU Dortmund University, 2014.
 - [33] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.
 - [34] O. Boykin, S. Ritchie, I. O’Connell, and J. Lin. Summingbird: A framework for integrating batch and online MapReduce computations. *VLDB Journal*, 7(13):1441–1451, 2014.
 - [35] V. Breazu-Tannen, P. Buneman, and L. Wong. *Naturally embedded query languages*. Springer, 1992.
 - [36] V. Breazu-Tannen and R. Subrahmanyam. *Logical and computational aspects of programming with sets/bags/lists*. Springer, 1991.
 - [37] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1), 2010.
 - [38] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 2011.
 - [39] 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.
 - [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. *VLDB Journal*, 8(4):401–412, 2014.
 - [42] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, 2003.

Bibliography

- [43] M. Charikar, L. O’Callaghan, and R. Panigrahy. Better streaming algorithms for clustering problems. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC. ACM, 2003.
- [44] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD*, 26(1), 1997.
- [45] S. Chaudhuri and V. Narasayya. TPC-D data generation with skew.
- [46] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. *SIGMOD*, 2000.
- [47] Y. Chen, J. Dunfield, and U. Acar. Type-directed automatic incrementalization. In *PLDI*, 2012.
- [48] Y. Chen and L. Tu. Density-based clustering for real-time stream data. *KDD*, 2007.
- [49] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR*, 2003.
- [50] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4), 2012.
- [51] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. Theory of privacy and anonymity. In M. Atallah and M. Blanton, editors, *Algorithms and Theory of Computation Handbook (2nd edition)*. CRC Press, 2009.
- [52] 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.
- [53] S. Consolvo, I. E. Smith, T. Matthews, A. LaMarca, J. Tabert, and P. Powledge. Location disclosure to social relations: Why, when, & what people want to share. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI, 2005.
- [54] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1), Apr. 2005.
- [55] S. Das, Y. Sismanis, K. Beyer, R. Gemulla, P. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.
- [56] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.
- [57] C. Demetrescu. Fully dynamic algorithms for path problems on directed graphs, 2001.

-
- [58] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. FOCS, 2000.
- [59] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *ACM*, 51(6), Nov. 2004.
- [60] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.
- [61] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1), 2007.
- [62] J. Dittrich, B. Seeger, D. Taylor, and P. Widmayer. Progressive merge join: a generic and non-blocking sort-based join algorithm. In *VLDB*, 2002.
- [63] J. Dittrich, B. Seeger, D. Taylor, and P. Widmayer. On producing join results early. In *PODS*, 2003.
- [64] T. Do and H. Gunawi. The case for limping-hardware tolerant clouds. In *HotCloud*, 2013.
- [65] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. SOCC, pages 14:1–14:14. ACM, 2013.
- [66] S. Dolan. Fun with semirings: a functional pearl on the abuse of linear algebra. In *ICFP '13*. ACM.
- [67] P. Domingos and G. Hulten. Mining high-speed data streams. KDD, 2000.
- [68] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *TOMS*, 16(1), 1990.
- [69] M. J. Egenhofer. Toward the semantic geospatial web. In *Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems*, GIS, 2002.
- [70] I. P. Egwuotuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.*, 65(3), Sept. 2013.
- [71] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *HPDC*, 2010.
- [72] E. Elnikety, T. Elsayed, and H. E. Ramadan. ihadoop: Asynchronous iterations for mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM, 2011.

Bibliography

- [73] EPFL. SQUALL. <https://github.com/epfldata/squall>.
- [74] D. Fabregat-Traver and P. Bientinesi. A Domain-Specific Compiler for Linear Algebra Operations. *VECPAR '12*, 2012.
- [75] M. Fire, D. Kagan, and R. Puzis. Data mining opportunities in geosocial networks for improving road safety. In *Electrical and Electronics Engineers in Israel*, (IEEEI), 2012.
- [76] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *PLDI '93*, 1993.
- [77] J. a. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Comput*, 46(4), Mar. 2014.
- [78] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *ICDE*, pages 231–242. IEEE, 2011.
- [79] J. R. Gilbert, S. Reinhardt, and V. B. Shah. High-performance graph algorithms from parallel sparse matrices. In *PARA*, 2007.
- [80] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*, pages 17–30, 2012.
- [81] A. Gounaris, N. Paton, A. Fernandes, and R. Sakellariou. Adaptive query processing: A survey. In *British National Conference on Databases*, 2002.
- [82] A. Gounaris, E. Tsamoura, and Y. Manolopoulos. Adaptive query processing in distributed settings. *Advanced Query Processing*, 36(1), 2012.
- [83] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.
- [84] J. L. Gross, J. Yellen, and P. Zhang. *Handbook of Graph Theory, Second Edition*. 2nd edition, 2013.
- [85] X. Gu, P. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *ICDE*, 2007.
- [86] S. Guha and A. McGregor. Approximate quantiles and the order of the stream. *PODS*, 2006.
- [87] A. Gupta and I. Mumick. *Materialized Views*. MIT Press, 1999.
- [88] B. Gutenberg and R. Richter. Frequency of earthquakes in california. *Bulletin of the Seismological Society of America*, 1944.

-
- [89] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.
- [90] C. Harrison, B. Eckman, R. Hamilton, P. Hartswick, J. Kalagnanam, J. Paraszczak, and P. Williams. Foundations for smarter cities. *IBM*, 54(4), 2010.
- [91] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2), 2000.
- [92] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *SIGMOD*, 1997.
- [93] J. M. Hernández-Muñoz, J. B. Vercher, L. Muñoz, J. A. Galache, M. Presser, L. A. H. Gómez, and J. Pettersson. The future internet. chapter Smart Cities at the Forefront of the Future Internet. Springer-Verlag, 2011.
- [94] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *ACM*, 48(4), July 2001.
- [95] Hortonworks. How big data is revolutionizing fraud detection in financial services. <https://hortonworks.com/blog/how-big-data-is-revolutionizing-fraud-detection-in-financial-services/>.
- [96] P. Indyk and D. Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC, 2005.
- [97] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.
- [98] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Sigmod*. ACM, 1990.
- [99] Y. E. Ioannidis, D. L. Lee, and R. T. Ng, editors. *ICDE*, 2009.
- [100] 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.
- [101] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.
- [102] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. In *SIGMOD*, 2008.
- [103] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

Bibliography

- [104] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2):175 – 193, 1984.
- [105] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.
- [106] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS, 1999.
- [107] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *VLDBJ*, 2014.
- [108] R. Kotto-Kombi, N. Lumineau, P. Lamarre, and Y. Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.
- [109] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. Franklin, and M. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, 2013.
- [110] 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.
- [111] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in mapreduce applications. In *Open Cirrus Summit*, 2011.
- [112] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in MapReduce applications. In *SIGMOD*, 2012.
- [113] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *OSDI '12*.
- [114] D. Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, 2001.
- [115] B. Li, Y. Diao, and P. Shenoy. Supporting scalable analytics with latency constraints. *VLDB*, 8(11):1166–1177, 2015.
- [116] 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.
- [117] B. Liu, M. Jbantova, and E. Rundensteiner. Optimizing state-intensive non-blocking queries using run-time adaptation. In *ICDE Workshop*, 2007.

-
- [118] 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.
- [119] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB* '12, 2012.
- [120] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.
- [121] E. Lu and R. Hamilton. Avalanches of the distribution of solar flares. *Astrophysical Journal*, 1991.
- [122] D. Maier. *Theory of Relational Databases*. Computer Science Press, 1983.
- [123] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
- [124] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
- [125] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases, VLDB*, 2002.
- [126] A. Marathe and K. Salem. Query processing techniques for arrays. *VLDBJ*, 11(1), 2002.
- [127] N. Marz. STORM: Distributed and fault-tolerant realtime computation. <https://github.com/nathanmarz/storm>.
- [128] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.
- [129] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.
- [130] E. Meijer. The world according to LINQ. *Communication ACM*.
- [131] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1), 2016.
- [132] S. Mihaylov, Z. Ives, and S. Guha. REX: Recursive, delta-based data-centric computation. *PVLDB*, 5(11), 2012.

Bibliography

- [133] 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.
- [134] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, 2005.
- [135] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.
- [136] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, 2013.
- [137] G. Neukum and B. Ivanov. *Crater size distributions and impact probabilities on Earth from lunar, terrestrial planet, and asteroid cratering data*. 1994.
- [138] M. Newman. Power laws, pareto distributions and zipf’s law. *Contemporary Physics*, 2005.
- [139] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. *SOSP ’13*, 2013.
- [140] M. Nikolic, M. ElSeidy, and C. Koch. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *SIGMOD ’14*.
- [141] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, 2011.
- [142] F. Olken. Random sampling from databases, 1993. PhD Thesis, UC Berkeley.
- [143] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.
- [144] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *Annual Technical Conference. USENIX*, 2008.
- [145] N. Paton, J. Buenabad, M. Chen, V. Raman, G. Swart, I. Narang, D. Yellin, and A. Fernandes. Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. *VLDBJ*, 18(1), 2009.
- [146] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.
- [147] PostgreSQL Global Development Group. PostgreSQL. <http://www.postgresql.org>.
- [148] D. Price. Networks of scientific papers. In *Science*, 1965.

-
- [149] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.
- [150] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.
- [151] F. R. Sayed and M. H. Khafagy. SQL TO Flink Translator. *International Journal of Computer Science Issues*, 12(1), January 2015.
- [152] S. Z. Sbz, S. Zdonik, M. Stonebraker, M. Cherniack, U. Etintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26, 2003.
- [153] H. Schaffers, N. Komninos, M. Pallot, B. Trousse, M. Nilsson, and A. Oliveira. The future internet. chapter Smart Cities and the Future Internet: Towards Cooperation Frameworks for Open Innovation. 2011.
- [154] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD*, 1989.
- [155] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2002.
- [156] A. F. Simpao, L. M. Ahumada, J. A. Gálvez, and M. A. Rehman. A review of analytics and clinical informatics in health care. *J. Med. Syst.*, 38(4), 2014.
- [157] M. P. Singh, M. A. Hoque, and S. Tarkoma. Analysis of systems to process massive data stream. *CoRR*, 2016.
- [158] Slick: Functional relational mapping for scala. <http://slick.lightbend.com/>.
- [159] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. *CGO '14*, pages 23:23–23:32. ACM, 2014.
- [160] D. G. Spampinato and M. Püschel. A basic linear algebra compiler for structured matrices. In *CGO '16*, pages 117–127. ACM, 2016.
- [161] J. Stamos and H. Young. A symmetric fragment and replicate algorithm for distributed joins. *Transactions on Parallel and Distributed Systems*, 4(12), 1993.
- [162] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *VLDB*, 2001.
- [163] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all? Part 2: Benchmarking results. In *CIDR*, 2007.

Bibliography



- [164] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD*, 34(4), Dec. 2005.
- [165] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *VLDB*, 8(11), 2015.
- [166] 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.
- [167] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.
- [168] 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.
- [169] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.
- [170] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD*, 2011.
- [171] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), 2000.
- [172] P. Vagata and K. Wilfong. Scaling the Facebook data warehouse to 300 PB. <https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/>, 2014.
- [173] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *EuroSys*, 2013.
- [174] S. Venkataraman, I. Roy, A. AuYoung, and R. Schreiber. Using R for iterative and incremental processing. In *HotCloud*, 2012.
- [175] A. Vitorovic. *Squall: Scalable Real-time Analytics using Efficient, Skew-resilient Join Operators*. PhD thesis, EPFL, 10 2016.
- [176] A. Vitorovic, M. Elseidy, and C. Koch. Load balancing and skew resilience for parallel joins. Technical Report 203656, EPFL, 2015.
- [177] C. Walton, A. Dale, and R. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *VLDB*, 1991.
- [178] S. Wang and E. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. In *EDBT*, 2009.

-
- [179] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *PPSC*, 1999.
 - [180] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Parallel and Distributed Information Systems*, 1991.
 - [181] Wolfram Research, Inc. Mathematica 8.0.
 - [182] Y. Xing, S. Zdonik, and J. Hwang. Dynamic load distribution in the Borealis stream processor. In *ICDE*, 2005.
 - [183] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP Algorithms. PLDI '01.
 - [184] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel DBMS, 2009.
 - [185] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, 2008.
 - [186] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.
 - [187] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.
 - [188] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.
 - [189] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. *SOSP*, 2013.
 - [190] 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.
 - [191] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using MapReduce. *VLDBJ*, 5(11), 2012.
 - [192] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-efficient numerical computing without SQL. In *CIDR*, 2009.
 - [193] Y. Zhou, B. Ooi, and K. Tan. Dynamic load management for distributed continuous query systems. In *ICDE*, 2005.
 - [194] G. Zipf. Human behaviour and the principle of least effort. In *Addison-Wesley, Reading, MA*, 1949.

Mohammed ElSeidy

✉ Avenue de la gare 38
1022 Chavannes, Switzerland

✉ mohammed.elseidy@epfl.ch
☎ +41 78 714 76 03

 [linkedin.com/in/mohamed-elseidy](https://www.linkedin.com/in/mohamed-elseidy)
 github.com/elseidy

Education

- 2011 - Today PhD. in the field of Database Systems, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (**EPFL**), Switzerland. PhD thesis advisor: [Prof. Christoph Koch](#)
- 2009 - 2011 MSc. Thesis in Computer Science, King Abdullah University for Science and Technology (**KAUST**), KSA, GPA: 3.78/4.0. M.Sc. thesis advisor: [Dr. Panos Kalnis](#)
- 2004 - 2009 BSc. in Computer Science Engineering and Automatic Control, **Alexandria University**, Egypt, GPA: 3.8/4.0

Core Experience

- 2011-Present **DATA lab at EPFL**, Lausanne, Switzerland, Doctoral Assistant, and worked on the following projects:
- ✦ **Squall** (github.com/epfldata/squall): is an online distributed SQL-query processing engine (e.g., Hive) built on top of Twitter's Storm. I am one of the **core** developers of Squall which is a Java **open-source** project with more than 50K loc. I am responsible for leading the project, developing the core backbone, and designing approaches that achieve scalable load balance by up to a factor of **80×**.
 - ✦ **Lago**: is a compiler framework for linear algebra programs dedicated for machine learning, graph algorithms, and statistical analysis. I am one of the **main** developers. I am responsible for the design and development of the framework. It generates optimized programs up to **10 – 100×** better in performance.
 - ✦ **Linview**: is a compiler framework that transforms linear algebra programs into efficient update triggers optimized for incremental computation. I am one of the **main** developers. My contributions present techniques that outperform current approaches by **100 – 1000×**.
 - ✦ I worked on various Big Data projects during my PhD and I served as a teaching assistant for the **Big Data** and **Maths** courses which include more than 120 students each. I lead large projects and teams of master students. Some of these groups, were awarded best project awards such as Wikilynx and **Submetrics** which was featured on **MediaCom**.
-
- 2014 **Microsoft Research**, Silicon Valley, California, USA (3-month internship, June – August 2014)
- Intern in the **Coconut Project**, a symbolic computation compiler for machine learning and scientific programs. I was responsible for building the core optimizer for the compiler using Monte Carlo Tree search, which improved the performance of generated programs by up to **10×**.
-
- 2010 **Technische Universität München**, Munich, Germany (3-month internship, July – September 2010)
- As a research assistant, I was responsible for developing and evaluating parallel linear algebra kernels using IBM's **supercomputer** BLUE GENE/P. My programs achieved up to **2×** better performance.
-
- 2011 **KAUST**, Master Thesis, and worked on the following projects:
- ✦ **GraMi** (github.com/ElSeidy/GraMi): is a novel framework for frequent subgraph mining in large graphs that outperforms existing techniques by up to **1000×** in performance. I was **the main developer** of GraMi which is an open source Java project with around 10K loc. GraMi is extensively used in academia.
 - ✦ **Distributed Query Engine**: Participated in the international SIGMOD competition to design and develop a distributed database management system. I was **ranked 7th** worldwide.

Technical Skills

| | |
|-----------------------|--|
| Programming Languages | Java, C# (.NET), C/C++, Scala, SQL, HTML, JavaScript, CSS, CUDA, Python, MATLAB, Octave, R, Assembly |
| Database Systems | Relational: Oracle DB, MySQL, T-SQL, PostgreSQL / NoSQL: Cassandra, HBase, BerkeleyDB, MongoDB Very good understanding of general database design, optimization and configuration |
| Tools | Hadoop, Hive, Yarn, Spark, Storm, SVN, Git, Maven, LaTeX, JUnit, MPI, JDBC, BLAS, Intel MKL, Cordova/PhoneGap |
| Software Development | Scrum Master / Agile Coach |

Additional Experience

- 2009 **Globalimpact SW**, Alexandria, Egypt
Intern in a software company that develops desktop, mobile, and web applications. I was responsible for developing SQL plugins for Microsoft products including Powerpoint and Excel using C# and the .NET framework. The plugins allow connecting, querying, and editing Microsoft SQL server databases.
- 2007–2009 **Freelancer**, Egypt
Developing software for customers. The main products include Bluetooth phone-webcam, graph algorithms visualization, and an end-to-end RSA encryption framework.

Honors

- 2011 **EPFL Graduate Fellowship**: in recognition to my academic excellence.
- 2011 **KAUST Academic Excellence Award**: a financial award in acknowledgement to academic merit.
- 2009 **KAUST Graduate Fellowship**: one of the three students picked up from a large number of applicants from Africa and Asia.
- 2008 **Microsoft Innovation Center**: chosen as one of the three finalists to be granted an excellence award in which more than 5,000 Computer engineers participated in Egypt.
- 2004–2009 **Academic Excellence Recognition**: received bachelors distinction awards for 5 consecutive years.

Technical Interests

- Distributed Data Analytics and Machine Learning
- Scalable Online Stream Processing
- Distributed Systems and the Hadoop Ecosystem
- Automation Tools for DevOps


Personal Activities

- Valizo **Co-founder**, Seattle USA, September–December 2014
- Valizo is a phone connected smart suitcase that saves money, time, and effort. I was responsible for the research and development area of the product.
- Social Events Organizing and participating in social events, especially those that gather diverse cultures. I am interested in charity work and fund raising events.
- Learning skills I am always working on learning new skills in different areas. Currently I am studying economics, in particular crypto-economics, and swing-trading.
- Hobbies Boxing, gym, latin-dancing, reading

Speaking Languages

English: Fluent / Arabic: Mother Tongue / French: Basic proficiency, B1 equivalent

Publications

- PVLDB **Towards Incremental Computation of Advanced Analytics**, *Mohammed Elseidy, Amir Shaikhha, et al.*, 2017, Under Submission.
- PVLDB **Squall: Scalable real-time analytics**, *Aleksandar Vitorovic, Mohammed Elseidy et al.*, 2016.
- PVLDB **Scalable and adaptive online joins**, *Mohammed Elseidy, Abdallah Elguindy, Alexandar Vitorovic, Christoph Koch*, 2014.
- PVLDB **GraMi: Generalized frequent pattern mining in a single large graph**, *Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis*, 2014.
- SIGMOD **LINVIEW: Incremental view maintenance for complex analytical queries**, *Milos Nikolic, Mohammed ElSeidy, Christoph Koch*, 2014.
- The detailed list, including the rest of my publications, is available at  [Google Scholar](#)

