Efficient Incremental Data Analysis

THÈSE N^{O} 7183 (2016)

PRÉSENTÉE LE 19 AOÛT 2016

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

Milos NIKOLIC

acceptée sur proposition du jury:

Prof. V. Kuncak, président du jury
Prof. C. Koch, directeur de thèse
Prof. D. Olteanu, rapporteur
Dr J. Goldstein, rapporteur
Prof. V. Cevher, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2016

To Milena and Andrej with love

Acknowledgements

This Ph.D. dissertation would have never been possible without the support of my teachers, colleagues, friends, and family members.

Foremost, I would like to thank my advisor Prof. Christoph Koch for the guidance and mentorship during my Ph.D. journey. I am deeply indebted to Christoph for being an inspiring and fair mentor, for selflessly sharing his knowledge about research and academia, for his unceasing support and thoughtful advice that strengthened the quality of this research, for helping me develop critical thinking skills, and for offering me the opportunities I could only dream about.

I am thankful to my thesis committee members, Prof. Viktor Kuncak, Prof. Volkan Cevher, Dr. Jonathan Goldstein, and Prof. Dan Olteanu, for devoting their time and energy in evaluating my thesis and providing useful feedback. I would also like to thank Prof. Oliver Kennedy, Prof. Yanif Ahmad, and Prof. Johannes Gehrke for the fruitful collaborations at the beginning of my Ph.D. journey. Their advice, guidance, and feedback were extremely valuable and motivating for a then-young researcher. I am also very grateful to Jonathan Goldstein and Badrish Chandramouli for being supportive and inspiring mentors during my internship at Microsoft Research in Redmond and for encouraging me to explore new research directions.

I would like to thank all the current and former members of the DATA lab at EPFL, Aleksandar Vitorovic, Mohammed ElSeidy, Mohammad Dashti, Vojin Jovanovic, Yannis Klonatos, Daniel Lupei, Amir Shaikhha, Immanuel Trummer, Lionel Parreaux, Andrej Spielmann, Abdallah Elguindy, Thierry Coppey, and Andres Nötzli, who have provided not only valuable feedback and support on my research, but they have created a great work environment. I am also greatful to the external collaborators, Nitin Gupta, Sudip Roy, Gabriel Bender, and Lucja Kot, for a great and productive time working together. I owe special thanks to Simone Muller for helping me with the administrative procedures of EPFL and making me feel comfortable in a foreign country. Finally, I thank Simone and Lionel for translating my thesis abstract to French.

I am deeply grateful for support and encouragement from my friends and family. I dedicate this thesis to my wife Milena and son Andrej who motivated me to pursue my dreams since the very beginning of my Ph.D. studies. Thank you for tolerating my long working hours and stressful deadlines and for making my life much more special.

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

Lausanne, 7 August 2016 Milos Nikolic

Abstract

Many data-intensive applications require real-time analytics over streaming data. In a growing number of domains – sensor network monitoring, social web applications, clickstream analysis, high-frequency algorithmic trading, and fraud detections to name a few – applications continuously monitor stream events to promptly react to certain data conditions. These applications demand responsive analytics even when faced with high volume and velocity of incoming changes, large numbers of users, and complex processing requirements. Developing suitable online analytics engine that meets these requirements is challenging.

In this thesis, we study techniques for efficient online processing of complex analytical queries, ranging from standard database queries to complex machine learning and digital signal processing workflows. First, we focus on the problem of efficient incremental computation for database queries. We have developed a system, called DBToaster, that compiles declarative queries into high-performance stream processing engines that keep query results (views) fresh at very high update rates. At the heart of our system is a recursive query compilation algorithm that materializes a set of supporting higher-order delta views to achieve a substantially lower view maintenance cost. We study the trade-offs between single-tuple and batch incremental processing in local execution, and we present a novel approach for compiling view maintenance code into data-parallel programs optimized for distributed execution. DBToaster supports millions of complete view refreshes per second for a broad range of queries and outperforms commercial database and stream engines by orders of magnitude.

We also study the incremental computation for queries written as iterative linear algebra, which can capture many machine learning and scientific calculations. We have developed a framework, called LINVIEW, for capturing deltas of linear algebra programs and understanding their computational cost. Linear algebra operations tend to cause an avalanche effect where even very local changes to the input matrices spread out and infect all of the intermediate results and the final view, causing incremental view maintenance to lose its performance benefit over re-evaluation. We develop techniques based on matrix factorizations to contain such epidemics of change and make incremental view maintenance of linear algebra practical and usually substantially cheaper than re-evaluation. We show, both analytically and experimentally, the usefulness of these techniques when applied to standard analytics tasks.

Our last research question concerns the integration of general-purpose query processors and domain-specific operations to enable deep data exploration in both online and offline analysis. We advocate a deep integration of signal processing operations and general-purpose

query processors. We demonstrate that in-situ processing of tempo-relational and signal data through a unified query language empowers users to express end-to-end workflows more succinctly inside one system while at the same time offering orders of magnitude better performance than existing popular data management systems.

Keywords: incremental view maintenance (IVM), materialized views, databases, stream processing, batch processing, higher-order IVM, distributed IVM, DBToaster, incremental linear algebra, compilation, signal processing

Résumé

De nombreuses applications faisant un usage intensif de données nécessitent des analyses en temps réel sur ces lues en continu (streaming). Dans un nombre croissant de domaines – surveillance de réseaux de capteurs, applications web sociales, analyse de flux de clics, trading algorithmique à haute fréquence et détection de fraudes, pour n'en citer que quelques-uns – les applications surveillent en permanence les événements présents dans les flux afin de réagir rapidement à certaines conditions. Ces applications exigent des analyses réagissant promptement, même lorsqu'elles sont confrontées à un volume et à une fréquence élevés de changements entrants, à un grand nombre d'utilisateurs, ainsi qu'à des exigences de traitement complexes. Développer un moteur d'analyse en ligne approprié qui répond à ces exigences est difficile.

Dans cette thèse, nous étudions des techniques efficaces de traitement en ligne de requêtes analytiques complexes, allant de requêtes de base de données standard à des systèmes complexe d'apprentissage automatique et de traitement des signaux numériques. Tout d'abord, nous nous concentrons sur le problème du calcul incrémental efficace des requêtes de base de données. Nous avons développé un système, appelé DBToaster, qui compile les requêtes déclaratives et produit des moteurs de traitement de flux haute performance qui maintiennent les résultats des requêtes (vues) à des taux de mise à jour très élevés. Au cœur de notre système, un algorithme de compilation de requêtes récursif matérialise un ensemble support de deltas d'ordre supérieur pour atteindre un coût de maintenance des vues considérablement réduit. Nous étudions les compromis entre le traitement incrémental tuple-par-tuple et par lots dans le contexte d'une exécution locale, et nous présentons une nouvelle approche pour la compilation de code de maintenance de vues vers des programmes aux données parallèles optimisés pour une exécution parallèle distribuée. DBToaster permet des millions de rafraîchissements complets de vues par seconde pour une large gamme de requêtes, et surpasse les moteurs de base de données et de flux commerciaux par plusieurs ordres de magnitude.

Nous étudions également le calcul incrémental pour les requêtes en algèbre linéaire itérative, qui peut représenter beaucoup de calculs scientifiques et d'apprentissage automatique. Nous avons élaboré un cadre, appelé Linview, pour capturer les deltas des programmes d'algèbre linéaire et la compréhension de leur coût de calcul. Les opérations d'algèbre linéaire ont tendance à provoquer un effet d'avalanche où même des changements très locaux aux matrices d'entrée infectent tous les résultats intermédiaires et la vue finale, rendant le maintien incrémental des vues moins performant que la réévaluation. Nous développons des techniques

basées sur la factorisation de matrice pour contenir de telles épidémies de changement et rendre le maintien incrémental des vues de type algèbre linéaire pratique et généralement nettement moins cher que la réévaluation. Nous montrons, à la fois analytiquement et expérimentalement, l'utilité de ces techniques lorsqu'elles sont appliquées à des tâches d'analyse standard.

Notre dernière question de recherche concerne l'intégration des processeurs de requête à usage général et des opérations spécifiques à un domaine pour permettre l'exploration de données en profondeur à la fois d'analyse en ligne et hors ligne. Nous préconisons une intégration profonde des opérations de traitement de signaux et processeurs de requête à usage général. Nous démontrons que in situ le traitement des données tempo-relationnelles et de signaux à travers un langage de requête unifiée permet aux utilisateurs d'exprimer des flux de travail de bout en bout plus succinctement à l'intérieur d'un système, tout en offrant en même temps des ordres de grandeur de meilleures performances que la gestion des données populaires des systèmes existants.

Mots clés : vue incrémentale maintenance (IVM), vues, bases de données, traitement des flux, traitement par lots, IVM supérieur, IVM distribué, DBToaster, algèbre linéaire incrémentiel, compilation, traitement du signal matérialisé.

Contents

List of Figures

List of Tables

1 Introduction

In this thesis, we study techniques for building systems for efficient online processing of complex analytical queries, ranging from standard database queries to complex machine learning and digital signal processing workflows.

Sophisticated data analysis plays an essential role in today's business world. Many companies collect and analyze large volumes of data to better understand their business processes, improve customer service, and, ultimately, generate more profit. For many decades, business intelligence has been relying on after-the-fact exploration in traditional data warehouses. With the pace of business constantly accelerating, companies have started now focusing on providing more responsive analytics involving complex forms of mining and learning from data in order to stay competitive in a global marketplace.

Developing suitable responsive analytics engine is challenging. Computational problems in many domains often have to process astonishing volumes of generated data. For instance, Facebook manages a data warehouse storing around 300PB of data and having an incoming daily rate of 600TB [11]; the Large Hadron Collider at CERN generates around 30PB of raw event data per year used by physicists in scientific simulations [3]. Data-intensive processing of any kind over these big working sets demands massively scalable solutions. Recently, many frameworks for scalable data processing have emerged: most notably MapReduce [62] and Spark [152, 151] for general-purpose data-parallel processing, along with specialized systems for large-scale processing of structured data [33, 143, 121], array processing [139, 58, 41, 35, 157], high-performance computing [65, 7], and machine learning and data mining [82, 154, 68, 100, 2, 110]. All these solutions primarily focus on efficiently processing large volumes of data.

Modern applications have to deal with not just big but also rapidly changing datasets. In a growing number of domains – Internet of Things, clickstream analysis, algorithmic trading, network monitoring, and fraud detection to name a few – applications compute real-time analytics over streams of continuously arriving data. Online and responsive analytics allow data miners, analysts, and statisticians to react promptly to certain, potentially complex, conditions in the data; or to gain preliminary insights from approximate or incomplete results

at very early stages of the computation. Existing tools for large-scale data analysis often lack support for dynamic datasets. High data velocity and complex processing requirements often force application developers to build ad hoc solutions to pull off maximum performance. More than ever, data analysis requires efficient, expressive, and scalable solutions to cope with the ever-increasing volume and velocity of generated data.

1.1 Requirements for Online Processing Systems

We identify three essential requirements for online processing systems to serve these modern applications: low-latency processing, support for long-running queries with complex logic, and scalable behavior. We describe them in more detail next.

Low-latency Incremental Processing Most datasets evolve through changes that are small compared to the overall dataset size. For example, the Internet activity of a single user, like her clickstream logs or online shopping history, represents only a tiny portion of the collected data. Providing up-to-date analytical results by recomputing from scratch on every (moderate) change is almost always inefficient.

These observations motivate *incremental data analysis*. Incremental processing combines the previously computed result with incoming changes to express the difference in the final result. Intuitively, small input changes often cause small output changes, which motivates us to use computationally cheaper methods for updating the results instead of re-evaluating everything from scratch. In most real-time applications, incremental computation is critical for online systems to sustain high update rates and achieve low latency.

Support for Complex Continuous Queries[1] Online processing systems use continuous (long-running) queries to analyze dynamic datasets. Two desirable properties of such query languages are 1) expressiveness – users want to ask queries that can capture complex conditions in streaming data, and 2) declarativity – users want to specify queries using high-level operators rather than low-level programming models.

Traditional online processors provide relational operators (e.g., selection, projection, join, etc.) and SQL-like query syntax for expressing grouped aggregations over streaming data. In the quest for better insights, modern applications increasingly demand more powerful analytics, like SQL queries with nested aggregates or sophisticated algorithms from domains like machine learning, data mining, scientific computing, and digital signal processing. Data processing systems supporting both relational and domain-specific operations can greatly empower users to perform complex data analysis. However, the challenge remains how to execute such workflows efficiently over evolving datasets.

Scalable Processing Emerging data-intensive applications demand scalable online systems for querying and managing large datasets. The motivation for using these systems may vary

[1] We will, throughout this thesis, use stream processing and continuous queries interchangeably. We will not aim to ensure bounded state size by window semantics, unless otherwise stated.

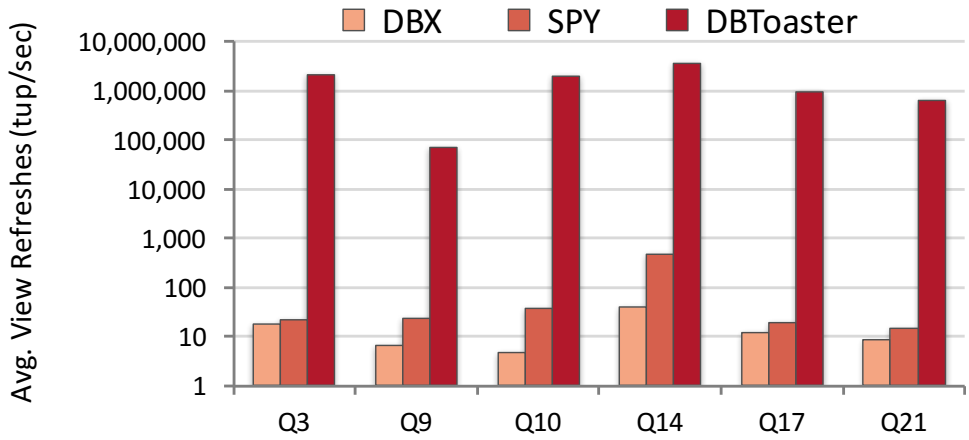


Figure 1.1 – Performance comparison of a commercial database system (DBX), a stream processing engine (SPY), and DBToaster on maintaining the results of a set of TPC-H queries for a stream of single-tuple updates to the base relations. Higher numbers are better.

depending on the application domain. In applications managing large stateful working sets, like real-time data warehouses, distributed execution aims to speed up query evaluation and accommodate growing memory requirements. Scalable behavior is also necessary for applications handling huge numbers of concurrent users; for instance, increasingly many IoT applications need to run the same query logic over a large collection of signals from sensor devices, and scaling out their processing is essential.

In this thesis, our primary focus is on achieving these critical requirements when building systems for efficient online analysis of relational, tempo-relational, and array data. Sometimes our solutions also provide other desirable features, like fault-tolerance in distributed execution, but studying them in more detail is out of the scope of this work.

1.2 Limitations of Existing Systems

Existing data processing systems often fail to simultaneously address the above requirements. Traditional relational databases offer rich query support but focus on high throughput rather than low latency. Many commercial implementations support incremental view maintenance – a technique for efficiently refreshing the contents of materialized views upon changes in base relations [75, 51, 95]. These systems, however, support only restricted classes of SQL queries, namely only flat queries but not queries with nested aggregates [9, 4]. Conventional data stream processing systems [112, 13, 32, 18] also exploit incremental computation to reduce the work over overlapping windows of input data. Their usefulness is yet limited by their window semantics, inability to handle long-lived data, and lack of support for complex queries. Both traditional databases and stream processing engines have limited scalability.

Example 1.2.1 Relational databases and stream processing engines support incremental processing but often fail to meet the performance requirements of modern real-time applications. To demonstrate that claim, we evaluate commercial implementations of a database system and a stream processing engine using an application that continuously monitors streaming data. The application computes online analytics using a set long-running queries derived from the TPC-H benchmark – a standard benchmark for evaluating databases – and ranging from flat queries with multi-way joins to complex queries with nested aggregates. We synthesize a stream of insertions and deletions to the base relations that ensures a roughly fixed working set size of 30,000 tuples. Our goal is to maintain the result (materialized view) after each insertion or deletion.

Figure 1.1 shows the measured performance of these commercial systems expressed as view refreshes per second. Notice that the y-axis has a logarithmic scale and higher values correspond to better performance. We observe that these two systems can process up to 100 tuples per second while maintaining views constantly fresh after every update. These numbers are far away from the requirements of real-time applications that often need to sustain much higher stream rates, like hundreds of thousands of updates per second, and keep views fresh at microsecond latencies. There are several reasons for such poor performance of the benchmarked systems: both cannot incrementally maintain SQL queries with nested aggregates and default to expensive re-evaluation in such cases; the stream engine is designed to work with window semantics and is inefficient in handling long-lived data; finally, when processing small-sized updates, the database system incurs significant overheads from components not related directly to view maintenance (e.g., input parsing, logging, concurrency control, etc.). □

Scalable stream processing platforms offer low-level programming models that put the burden of expressing complex query plans on the application developer [28, 102, 117]. Others provide declarative continuous queries but with no efficient support for the incremental computation of complex analytics, like queries with nested aggregates [114, 45, 29, 153]. Scalable batch processing systems built around MapReduce and Spark aim for high throughput rather than low latency [121, 143, 33]. Their design favors infrequent bulk updates during which these systems are typically unresponsive.

More complex forms of data processing often take the form of linear transformations of vectors and matrices. Popular statistical environments, like MATLAB and R, offer high-level abstractions that simplify programming but lack the support for scalable full-dataset analytics. Simulating multidimensional array computations on top of traditional relational databases often results in poor performance [139]. Specialized systems for array processing [139, 41, 35, 108] provide scalable offline analysis of multidimensional array data but lack the support for incremental computation.

The database community has recognized the need for a tighter integration of data management systems and domain-specific algorithms. Numerical computing environments like MATLAB and R are unsuitable for general-purpose processing involving relational opera-

tions such as joins, filtering, or group-by aggregation. To enrich processing workflows with specialized routines, increasingly many data management systems integrate with statistical frameworks [39, 139, 58, 41, 152, 151, 147, 13]. However, the existing integration mechanisms are suboptimal performance-wise as they treat both sides as independent systems with fundamentally different data models and expensive intercommunication. Such loose system coupling is particularly unsuitable for real-time and incremental processing.

1.3 Contributions

In this thesis, we study the foundations, algorithms, and architectures of data management tools designed for large datasets that evolve rapidly through high-rate update streams. We develop techniques for low-latency incremental processing of complex queries within the context of three different systems. We present the main contributions of our work next.

1.3.1 Incremental View Maintenance for Database Queries

The core of this thesis studies the problem of efficient incremental computation of database queries. Our research builds upon a novel incremental processing technique, called the viewlet transform [93, 94, 95]. This technique materializes the query result and a supporting set of higher-order delta views to achieve a substantially lower view maintenance cost and eliminate certain expensive query operations, such as joins.

In this thesis, we present the lessons learned in an effort to make the viewlet transform practical and to understand its strengths and drawbacks. There are cases (inequality joins and certain nesting patterns) when a naïve viewlet transform is too aggressive, and certain parts of queries are better re-evaluated than incrementally maintained. We develop heuristic rules for trading off between materialization and lazy evaluation for the best performance.

We have built the DBToaster compilation framework, which implements the viewlet transform and transforms declarative database queries into high-performance stream processing engines that keep query results (views) fresh at very high update rates. Figure 1.1 shows that DBToaster can achieve up to 5 orders of magnitude higher view refresh rates than the state-of-the-art relational database and stream processing engines. We provide a thorough experimental evaluation of our system for a wide range of analytical SQL queries later on.

Support for complex queries DBToaster aims to combine the advantages of database systems (rich queries over recent and historical data, without restrictive window semantics) and stream processing engines (low latency and high view refresh rates). To achieve this goal, we develop techniques for efficient incremental computation of database queries, including those with nested aggregates for which no commercial implementation of incremental view maintenance exists.

Program specialization We argue that efficient view maintenance requires specialization of incremental programs. Data management systems incur significant overheads due to their complexity and the use of generic data structures and algorithms [95, 92, 116], which makes them inappropriate for low-latency processing of frequent updates, as demonstrated in Figure 1.1. We observe that nowadays most applications have static query workloads with template-derived queries. Knowing the workload in advance allows us to tailor query processing based on the application requirements and avoid unnecessary features of database systems. In this work, we present techniques for specializing incremental programs into low-level native or interpreted code and generating custom data structures to facilitate efficient maintenance operations.

Single-tuple vs. batch processing We study the trade-offs between tuple-at-a-time and batch incremental view maintenance in local settings. The former can yield simpler maintenance code, for instance, we can eliminate loops around input changes knowing they are of constant size. The latter can have positive or negative impacts on cache locality. We identify the cases when batch processing can significantly reduce view maintenance costs. But, we also demonstrate in our experiments that maintenance programs specialized for single-tuple processing can outperform generic batch implementations in surprisingly many cases, in almost one-half of the benchmarked queries. These results refute the widespread belief that batching always wins over tuple-at-a-time processing [122].

Distributed execution To cope with the ever-increasing volume and velocity of data, we develop techniques for distributed incremental view maintenance of database queries. This problem is significantly harder than that of distributed query optimization [98, 129, 20, 47] because we aim to parallelize view maintenance programs consisting of multiple update statement rather than individual queries. We build a framework for transforming local programs into data-parallel processing tasks running over a large-scale cluster. The framework consists of a set of simplification and heuristics rules for minimizing network communication and synchronization during distributed execution. Our distributed view maintenance implementation for standard database queries can deliver few-second latencies of processing tens of million of tuples using hundreds of workers.

1.3.2 Incremental Linear Algebra

In this thesis, we also study the incremental view maintenance problem for queries written as iterative linear algebra programs. Such queries can express complex data analyses, like machine learning algorithms or scientific calculations. The main challenge in incremental linear algebra is how to represent and propagate delta expressions that capture the difference between the new and old result. Linear algebra operations tend to cause an avalanche effect where even very local changes to the input matrices spread out and infect all of the intermediate results and the final view. Expressing such deltas naïvely and propagating them to subsequent statements quickly becomes more expensive than recomputing the entire program from scratch using the new input.

We develop techniques based on matrix factorizations to contain such epidemics of change. Our approach represent delta expressions in a *factored form*, as products of low-rank matrices, and utilizes a set of transformation rules to reshape linear algebra programs into their cheaper functional equivalents that are optimized for dynamic datasets. The factored form admits efficient incremental computation of linear algebra programs that involve expensive matrix operations, like full matrix-matrix multiplication and matrix inversion. We demonstrate both analytically and experimentally the efficiency of incremental computation on various fundamental data analysis methods, such as Ordinary Least Squares, batch gradient descent, PageRank, and matrix powers.

We have built LINVIEW, a compiler for incremental data analysis that exploits these novel techniques to generate efficient maintenance triggers. The compiler is easily extensible to couple with any underlying system that supports matrix manipulation primitives. We evaluate the performance of LINVIEW's generated code in both local and distributed settings. We show that incremental evaluation can orders of magnitude better performance than traditional re-evaluation.

1.3.3 Enabling Signal Processing over Data Streams

Our last research question concerns the integration of general-purpose query processors and domain-specific operations to enable deep data exploration in both online and offline analysis. As our motivating example, we consider Internet of Things applications that analyze sensor data coming from large networks of devices using queries that combine relational and signal processing operations. Reconciling these two seemingly disparate worlds of relational and signal (array) data, especially in the context of real-time analysis, is a challenging task.

In this thesis, we advocate a deep integration of domain-specific tools and general-purpose query processors. To demonstrate our approach, we have extended Trill [45] – a commercial stream processing engine based on the tempo-relational model – with signal processing functionality. We provide a unified query language that allows end-users to seamlessly interleave relational and signal operations when writing queries for online and offline analysis. In-situ processing of tempo-relational and signal data opens up the opportunity for incremental computation of entire workflows and avoids the performance overheads of existing loosely-coupled solutions. For domain experts, we provide frameworks exposing array abstractions for quick and easy integration of user-defined operators, like existing highly-optimized implementations, with the query language. Our deeply-integrated system can achieve orders of magnitude better performance than existing loosely-coupled data management systems on signal processing tasks in IoT scenarios.

1.4 Thesis Outline

This thesis is organized as follows. Chapter 2 introduces the concept of incremental view maintenance for database queries. Chapter 3 presents higher-order incremental processing. Chapter 4 and 5 describe the implementation of incremental view maintenance for database queries in local and distributed environments. Chapter 6 studies incremental computation of linear algebra programs. Chapter 7 studies the integration of signal processing and stream processing engines. Chapter 8 discusses related work, and Chapter 9 concludes this thesis.

This work includes material from several publications in which the author of this thesis is the lead author or a co-author. Chapter 2 presents material, namely the data model, the query language, and the idea of recursive incremental view maintenance, that was initially developed in previous work [93, 94, 25, 90] and later refined in the follow-up publications co-written by the author of this thesis [24, 95]. The author played a major role in designing and implementing the materialization heuristics, optimizations, and efficient evaluation strategies from Chapter 3, and in conducting experiments and discussing results from Chapter 4. These chapters combine material from our publications [95, 119]. The author also led the research and development of distributed incremental view maintenance [119] in Chapter 5, incremental linear algebra [120] in Chapter 6, and signal processing over stream data in Chapter 7.

2 Incremental View Maintenance

In this chapter, we introduce the concept of incremental view maintenance in databases. We present a state-of-the-art incremental view maintenance technique, called the viewlet transform, which we first describe informally through an example and then formalize using the previously defined query language. The content of this chapter includes material from our publications [95, 119], which refine the data model, the query language, and the initial idea of the viewlet transform from previous work [93, 94].

2.1 Concept: Incremental Computation in Databases

Database systems can pre-compute results of frequently asked queries to speed up their execution. Such stored representations, known as materialized views, require maintenance to keep their contents up to date for changes in base tables. Refreshing materialized views using recomputation is expensive for frequent and small-sized updates. In such cases, applying only incremental changes (deltas) to materialized views is usually more efficient than recomputing views from large base tables.

Due to their impact on performance, materialized views have become an import research topic in many domains (see the survey by Chirkova and Yang [51]), like traditional query processing [75, 49, 74, 38, 104, 131], stream processing [70, 78, 85, 21, 115], data mining [141, 81], and data warehousing [91, 48, 22, 149, 128, 79]. Materialized views are supported by most commercial database systems and tightly integrated into query optimization.

2.1.1 Classical Incremental View Maintenance

Incremental view maintenance (IVM) uses delta queries to capture the difference in the materialized contents caused by changes in base relations. Let $\langle Q, M(\mathbf{D})\rangle$ denotes a materialized view, where Q is the view definition query and $M(\mathbf{D})$ is the materialized contents for a given database $\mathbf{D}$. When the database changes from $\mathbf{D}$ to $(\mathbf{D}+\Delta\mathbf{D})$, where $\Delta\mathbf{D}$ represents insertions,

deletions, and updates to base tables, classical incremental view maintenance *evaluates* a delta query ΔQ to refresh $M(\mathbf{D})$.

$$M(\mathbf{D}+\Delta\mathbf{D}) = M(\mathbf{D}) + \Delta Q(\mathbf{D},\Delta\mathbf{D})$$

The delta ΔQ often has a simpler structure than Q (e.g., has fewer joins) and involves smaller delta updates instead of large base tables. So, computing ΔQ and refreshing $M(\mathbf{D})$ becomes cheaper than re-evaluating Q from scratch. Incremental view maintenance derives one delta query for each referenced base table. The derivation process relies on a set of change propagation rules defined for each operator of the view definition language. The derived delta query takes its role in the associated view maintenance trigger.

Example 2.1.1 Let query Q counts the tuples of a natural join of $R(A,B)$, $S(B,C)$, and $T(C,D)$ grouped by column B (i.e., for each distinct B value). Intuitively, we write the delta query for updates to R as:

$$\Delta_R Q := \mathrm{Sum}_{[B]}(\Delta R(A,B) \bowtie S(B,C) \bowtie T(C,D))$$

Let M_R, M_S, and M_T denote the materialized base tables R, S, and T, then the maintenance trigger for updates to R looks as follows.

Listing 2.1 Incremental view maintenance of Q for updates to R

```
1  ON UPDATE R BY ΔR:
2    M_R(A,B)  += ΔR(A,B)
3    ΔQ(B)     := Sum_[B](ΔR(A,B) ⋈ M_S(B,C) ⋈ M_T(C,D))
4    M_Q(B)    += ΔQ(B)
```

Here, `+=`, `:=`, and $\bowtie$ denote bag union, assignment, and natural join. The maintenance trigger applies ΔR to R and recomputes $\Delta_R Q$ to refresh M_Q. Under the standard assumption that $|\Delta R| \ll |R|$, incremental maintenance is cheaper than re-evaluation. □

Incremental view maintenance is often cheaper than naïve re-evaluation but is not free. Computing deltas can be expensive, like in Example 2.1.1, where $\Delta_R Q$ is a non-trivial join of one (small) input update and two (potentially large) base tables.

2.1.2 Idea: Recursive Incremental View Maintenance – The Viewlet Transform

Delta queries can be expensive despite their simpler form. For instance, a delta of an n-way join still references $(n-1)$ base tables. Instead of computing such a delta query from scratch, we could re-apply the idea of incremental processing to speed up the delta evaluation: store previously computed delta results, just as any other query result, and compute the delta of a delta query (second-order delta) to maintain the materialized delta result. That way, the second-order delta query maintains the first-order delta view, which in turn maintains the top-level view. Assuming that with each derivation deltas become simpler, we could recursively apply the same procedure until we get deltas with no references to base tables.

The described technique for constructing higher-order deltas is closer in spirit to discrete wavelet and numerical differentiation methods, and we use a superficial analogy to the Haar wavelet transform as the motivation for calling the base technique a *viewlet transform*. Here, we present just the intuition behind this technique and provide a more formal description later in this chapter.

The viewlet transform materializes the top-level view along with a set of auxiliary views that support each other's incremental maintenance. The materialization procedure starts from the top-level view and derives its delta queries for updates to base relations. For each delta query, the procedure materializes its update-independent parts such that the delta evaluation requires as little work as possible. In other words, it transforms $\Delta Q(\mathbf{D}, \Delta\mathbf{D})$ into an equivalent query $\Delta Q'$ that evaluates over a set of materialized views $M_1, \ldots, M_k$ and update $\Delta\mathbf{D}$:

$$\Delta Q(\mathbf{D}, \Delta\mathbf{D}) = \Delta Q'(M_1(\mathbf{D}), M_2(\mathbf{D}), ..., M_k(\mathbf{D}), \Delta\mathbf{D})$$

But note that $M_1, \ldots, M_k$ also require maintenance, which again relies on simpler materialized views. At first, it may appear counterintuitive that storing more data can reduce maintenance costs. However, the recursive incremental maintenance scheme makes the work required to keep all views fresh extremely simple. For flat queries, each individual aggregate value can be incrementally maintained using a constant amount of work [93, 94], which is impossible to achieve with classical incremental maintenance or re-evaluation.

Example 2.1.2 Let us apply recursive incremental view maintenance on the query of Example 2.1.1 and updates to R. Considering $\Delta_R Q$, we materialize its update-independent part $S(B,C) \bowtie T(C,D)$ as an auxiliary view $M_{ST}(B)$. We projected away C and D as they are irrelevant for the computation of $\Delta_R Q$. Repeating the same procedure for updates to T, we materialize $R(A,B) \bowtie S(B,C)$ as $M_{RS}(B,C)$ to facilitate computing of $\Delta_T Q$. For updates to S, we materialize $R(A,B) \bowtie T(C,D)$ separately as $M_R(B)$ and $M_T(C)$[1].

Next, we derive second-order deltas for M_{ST} and M_{RS}. Repeating the same delta derivation for updates to all three base relations, we materialize one additional view $M_S(B,C)$ representing the base relation S. Further derivation produces delta expressions with no base relations. Overall, recursive view maintenance materializes queries at three different levels: the top-level query M_Q, two auxiliary views M_{RS} and M_{ST}, and the base tables M_R, M_S, and M_T. The maintenance trigger for updates to R looks as follows.

Listing 2.2 Recursive incremental view maintenance of Q for updates to R

```
1  ON UPDATE R BY ΔR:
2    MQ(B)     += Sum[B](ΔR(A,B) ⋈ MST(B))
3    MRS(B,C)  += Sum[B](ΔR(A,B) ⋈ MS(B,C))
4    MR(B)     += Sum[B](ΔR(A,B))
```

We similarly build triggers for updates to S and T. □

[1] An efficient implementation of the viewlet transform avoids materializing query results with disconnected join graphs for performance reasons. Chapter 3 describes such optimizations in more details.

The viewlet transform can produce triggers with lower complexity than classical maintenance triggers. In the previous example, each statement performs at most one join between the delta relation and one materialized view, which is clearly less expensive than the classical approach. In general, if classical IVM is a good idea, then repeating it recursively is an even better idea. The same efficiency improvement argument in favor of IVM of the base query also holds for IVM of the delta query. Considering that joins are expensive and this approach simplifies or eliminates them, the viewlet transform has the potential for excellent query performance.

2.2 Data and Query Model

In this section, we introduce formalisms used for studying the problem of incremental view maintenance for relational queries. We present the internal data model, *generalized multiset relations* (GMRs), which enables a uniform treatment of different forms of updates (insertions and deletions) during incremental view maintenance. We define the query language, *AGgregate CAlculus* (AGCA), which consists of a few operators capable of expressing most of SQL and is amenable to powerful optimizations due to its simplicity.

Our data model generalizes multiset relations to collections of tuples where each tuple is annotated with a rational multiplicity (i.e., from $\mathbb{Q}$). As such multiplicities can be positive or negative, we can treat databases and updates as well as insertions and deletions uniformly – for instance, a deletion is a relation with negative multiplicities, and applying an update to a database means unioning/adding it to the database. In our model, such rational multiplicities can also keep (potentially non-integer) aggregate values of group-by queries, in contrast to SQL which stores these values in an additional column (thus, changing the query result schema). Maintaining aggregates in the multiplicities allows for simpler and cleaner bookkeeping in delta processing – for instance, growing an aggregate means changing the multiplicity of a tuple rather than deleting the tuple and inserting a tuple with the new aggregate value. Furthermore, we can associate multiple "multiplicities" ($\mathbb{Q}^k$) to a tuple to maintain multiple aggregates inside a single GMR.

Our query language (AGCA) consists of just four operations – addition, its inverse, multiplication, and sum-aggregation – constructed over GMRs and infinite interpreted relations (which capture conditions, such as $a < b$ and $x = 5$). AGCA is based on the ring-theoretic framework [93, 94] which defines the query language as a polynomial ring over GMRs with an addition operation that at once generalizes multiset union (as known from SQL) and updating, and a multiplication operation that generalizes the natural join operation. This syntactic simplicity of AGCA enables rich optimizations, as described in Chapter 3.

The query language implements sideways information passing and enforces range restriction (variable bindings) as known in the context of relational calculus. Supporting such bindings eliminates the need for an explicit selection operation, which AGCA encodes as a multiplication of a query with a condition (interpreted relation) just like in relational calculus. Multiplication is defined in such a way that query results are guaranteed to be always finite.

2.2.1 Data Model

We model a relation tuple $\vec{t}$ as a tuple of values with an associated schema, formally defined as a function from a vocabulary of column names $\text{dom}(\vec{t})$ to data values. We write $\vec{t}$ as $\langle A : v \mid A \in \text{dom}(\vec{t})\rangle$, where v is a value from the domain of column A. The set of all tuples is denoted by $\mathbb{T}$, and $\langle\rangle$ signifies the empty tuple.

We model a *generalized multiset relation (GMR)* as a collection of relation tuples, each annotated with a tuple of rational multiplicities. We define a GMR $R : \mathbb{T} \to \mathbb{Q}^k$ as a function from relation tuples to tuples of rational numbers such that $R(\vec{t}) \neq \langle 0\rangle^k$ for at most a finite number of tuples $\vec{t}$. A GMR encodes one relation with k aggregate columns (e.g., count, sum, etc.) where the rational multiplicities represent aggregate values. We write $\text{sch}(R)$ to denote a common schema of GMR R, which subsumes the tuple schemas. Below, we also use classical singleton relations (without multiplicities) and the natural join operator $\bowtie$. We write $\{\vec{t}\}$ to construct a singleton relation from tuple $\vec{t}$ with the schema $\text{sch}(\{\vec{t}\}) = \text{dom}(\vec{t})$. For tuples $\vec{s}, \vec{t}$ that are consistent ($\{\vec{s}\} \bowtie \{\vec{t}\} \neq \emptyset$), we can write $\vec{s}\vec{t}$ for the consistent concatenation ($\{\vec{s}\vec{t}\} = \{\vec{s}\} \bowtie \{\vec{t}\}$).

Example 2.2.1 Let R be a GMR of $\mathbb{Q}^3_{\text{Rel}}$

R	A	B	
	1	2	$\mapsto \langle c_1, s_1, a_1\rangle$
	3	5	$\mapsto \langle c_2, s_2, a_1\rangle$
	4	2	$\mapsto \langle c_3, s_3, a_1\rangle$

defined over column name vocabulary $\{A, B\}$, where c_i, s_i, and a_1 denote rational multiplicities (e.g., count, sum, and average aggregates). We only show entries with nonzero multiplicity. □

We denote the set of all GMRs with k-arity multiplicities by $\mathbb{Q}^k_{\text{Rel}}$. Without loss of generality, next we consider the query language over $\mathbb{Q}^2_{\text{Rel}}$ with the two fields storing the bag (count) multiplicity of a tuple (for bookkeeping purposes) and the aggregate value being computed. Our query language may be generalized from $\mathbb{Q}^2_{\text{Rel}}$ to $\mathbb{Q}^k_{\text{Rel}}$ for any $k > 2$ by cloning its behavior with respect to the "value" field. This generalization is omitted to avoid notation clutter.

2.2.2 Query Language

We now formally define AGCA over $\mathbb{Q}^2_{\text{Rel}}$. The language uses algebraic formulas to express queries (views) over generalized multiset relations. Valid queries result in relations with finite support, and because the tuples in a relation are unique, we can interpret query results as maps (dictionaries) with tuples being the keys and multiplicities being the values.

Syntax

AGCA expressions are built from constants, variables, relational atoms,conditions, and variable assignments ($:=$), using operations bag union $+$, natural join $*$, and aggregate sum $\text{Sum}_{\vec{A}}$. The abstract syntax is:

$$q ::= q * q \mid q + q \mid -q \mid c \mid x \mid R(\vec{t}) \mid \mathrm{Sum}_{\vec{A}}(q) \mid x\,\theta\,0 \mid x := q$$

Here x denotes variables (which we also call columns), $\vec{t}$ tuples of variables, $\vec{A}$ tuples of group-by variables, R relation names, c constants from $\mathbb{Q}$, and θ denotes comparison operators ($=$, $\neq$, $>$, $\geq$, $<$, and $\leq$). We also use $x\,\theta\,y$ as syntactic sugar for $(x-y)\,\theta\,0$.

Note that $x := q$ is a special condition essentially equivalent to $x = q$, with one catch. In relational calculus, both variables x and y are safe in $\phi \wedge x = y$ if at least one of them is safe in ϕ (the other variable can be *assigned* the value of the safe variable from ϕ). To make this information flow explicit, we create a syntactic distinction between the case where only one of the variables is safe from the left ($:=$) and the case where both are safe ($=$).

Informal Semantics

We first present a fragment of the language sufficient for expressing flat queries with aggregates.

- Relation $R(A_1, A_2, ...)$ represents the contents of a base table. It defines a mapping from every unique tuple of the relation to its multiplicity in $\mathbb{Q}^2$. The SQL equivalent is `SELECT A1, A2, ..., COUNT(*) FROM R GROUP BY A1, A2, ...`, where both multiplicity fields store the count aggregate.
- Bag union $q_1 + q_2$ merges tuples of q_1 and q_2, summing vector-wise their multiplicities.
- Natural join $q_1 * q_2$ matches tuples of q_1 with tuples of q_2 on their common columns, multiplying vector-wise their multiplicities.
- $\mathrm{Sum}_{\vec{A}}(q)$ serves as multiplicity-preserving projection. The result of $\mathrm{Sum}_{\vec{A}}(q)$ is the tuples of the projection of q on $\vec{A}$, and each tuple's multiplicity is the sum of the multiplicities of the tuples that were projected down to it. An aggregation $\mathrm{Sum}_{\vec{A}}R$ almost works like the SQL query `SELECT` $\vec{A}$`, SUM(1) FROM R GROUP BY` $\vec{A}$. The only difference is that SQL puts the aggregate values into a new column, while $\mathrm{Sum}_{\vec{A}}R$ puts them into the multiplicity of the group-by tuples. We can express more general aggregate summations using clever arithmetics on multiplicities, as shown below.
- Constant c can be interpreted as a singleton relation mapping the empty tuple to the multiplicity of $\langle 1, c\rangle$. Note that the empty tuple joins with any other tuple.
- Value term $f(x, y, \ldots)$, which generalizes variable x, is an interpreted relation defining a mapping from tuple $\langle x, y, \ldots\rangle$ to its multiplicity $\langle 1, f(x, y, \ldots)\rangle$. Constant c is a special case of a value term. Value terms are valid only if all variables are bound at evaluation time. For example, expression A has a free (unsafe) variable, thus it is invalid, while expression $R(A, B) * A$ has finite support. The latter expression corresponds to the SQL query `SELECT A, B, SUM(A) FROM R GROUP BY A, B`.
- Variable assignment $(x := v)$ lifts the aggregate multiplicities of value term v to tuple values. It defines a GMR with one column x and tuple values mirroring the aggregate

multiplicity field of v and having the multiplicity of $\langle 1,1\rangle$. Multiple variable assignments can be joined together to construct an arbitrary wide tuple.

- Comparison $(x\,\theta\,0)$ is an interpreted relation where each tuple has a multiplicity of either $\langle 0,0\rangle$ or $\langle 1,1\rangle$ depending on the truthfulness of the boolean predicate. Joining an expression with a comparison filters out tuples not satisfying the predicate by setting their multiplicity to $\langle 0,0\rangle$; the multiplicities of the matching tuples remain unchanged.

  The query language enforces a range restriction policy to ensure that all expressions define finite relations, as in relational calculus. That is, we may write $R * (A < B)$ for `SELECT A, B FROM R WHERE A < B`, without explicitly using a selection operation. However, $(A < B)$ by itself is not a valid query because an unbounded number of tuples satisfy the condition.

To support queries with nested aggregates, the query language generalizes the assignment operator to take on arbitrary expressions instead of just values. Then, variable assignment $(x := q)$ defines a finite-size relation containing tuples of expression q with non-zero count multiplicities extended by column x holding the aggregate multiplicities. Each output tuple has the multiplicity of $\langle 1,1\rangle$. Expression q may be correlated with the outside as usual in SQL. Variable assignment $(x := q)$ is a powerful and subtle operation which requires GMRs of type $\mathbb{Q}^2_{\text{Rel}}$ to maintain, separately, aggregates and true multiplicities.

With the above intuition in mind, the reader should be able to validate that the formal semantics shown in Figure 2.1 matches the given description. Note that in these semantics, the generalized union, join, and projection operations are denoted by $+$, $*$, and $\text{Sum}_{\vec{A}}$.

Semantics

We define the formal semantics of AGCA using an evaluation function $[\![\cdot]\!]$ that, for a query Q, a database $\mathbf{D}$, and a *context* — a tuple $\vec{b}$ of "bound variables" — evaluates to an element $[\![Q]\!](\mathbf{D},\vec{b})$ of $\mathbb{Q}^2_{\text{Rel}}$, as presented in Figure 2.1. Note that in several recursive cases arithmetic on the multiplicity/value tuples is performed vector-wise.

AGCA admits sideways information passing meaning that query expressions are evaluated relative to a given *context* $\vec{b}$, an association of variables and their values, provided from the outside. The query language, specifically the multiplication operation, dictates how such bindings are to be passed to the right during query evaluation.

The definition of $[\![R(\vec{x})]\!]$ allows column renaming. The evaluation of variables x (e.g., $[\![x]\!]$) *fails* if they are unbound at evaluation time. We consider a query in which this may happen illegal and exclude such queries from AGCA. Observe that $R-S = R+(-S)$ does not refer to the difference operation of relational algebra, but to the additive inverse for GMRs: for instance, $\emptyset - R = -R$ in AGCA ($\emptyset$ can be written in AGCA as the constant 0), while the syntactically same expression in relational algebra results in $\emptyset$. It is more appropriate to think of a GMR $-R$ as a deletion, where deleting "too much" results in a database with *negative tuples*.

Base Cases

Constant Value

$$[\![c]\!](\cdot,\cdot) := \vec{t} \mapsto \begin{cases} \langle 1, c\rangle \;..\; \vec{t} = \langle\rangle \\ \langle 0, 0\rangle \;..\; \text{otherwise} \end{cases}$$

Variable Value

$$[\![x]\!](\cdot,\vec{b}) := \vec{t} \mapsto \begin{cases} \textbf{fail} & ..\; x \notin \text{dom}(\vec{b}) \\ \langle 1, \vec{b}(x)\rangle & ..\; \text{otherwise, if } \vec{t} = \langle\rangle \\ \langle 0, 0\rangle & ..\; \text{otherwise} \end{cases}$$

Relation

$$[\![R(\vec{x})]\!](\mathbf{D},\vec{b}) := \vec{t} \mapsto \begin{cases} \langle m, m\rangle \;..\; m = R^{\mathbf{D}}(\langle A_i : \vec{t}(x_i) \mid A_i \in \text{sch}(R)\rangle), \\ \qquad\qquad \{\vec{b}\} \bowtie \{\vec{t}\} \neq \emptyset, |\text{dom}(\vec{t})| = |\text{sch}(R)| \\ \langle 0, 0\rangle \;..\; \text{otherwise} \end{cases}$$

Comparison

$$[\![x\,\theta\,0]\!](\mathbf{D},\vec{b}) := \vec{t} \mapsto \begin{cases} \textbf{fail} & ..\; x \notin \text{dom}(\vec{b}) \\ \langle 1, 1\rangle & ..\; \vec{b}(x)\,\theta\,0,\ \vec{t} = \langle\rangle \\ \langle 0, 0\rangle & ..\; \text{otherwise} \end{cases}$$

Recursive Cases

Bag Union

$$[\![Q_1 + Q_2]\!](\mathbf{D},\vec{b}) := \vec{t} \mapsto [\![Q_1]\!](\mathbf{D},\vec{b})(\vec{t}) +^{\mathbb{Q}^2} [\![Q_2]\!](\mathbf{D},\vec{b})(\vec{t})$$

Additive Inverse

$$[\![-Q]\!](\mathbf{D},\vec{b}) := \vec{t} \mapsto [\![Q]\!](\mathbf{D},\vec{b})(\vec{t}) *^{\mathbb{Q}^2} \langle -1, -1\rangle$$

Natural Join

$$[\![Q_1 * Q_2]\!](\mathbf{D},\vec{b}) := \vec{t} \mapsto \sum_{\substack{\{\vec{t}\}=\{\vec{r}\}\bowtie\{\vec{s}\} \\ \{\vec{b}\}\bowtie\{\vec{r}\}\neq\emptyset}}^{\mathbb{Q}^2} [\![Q_1]\!](\mathbf{D},\vec{b})(\vec{r}) *^{\mathbb{Q}^2} [\![Q_2]\!](\mathbf{D},\vec{b}\vec{r})(\vec{s})$$

Sum with Group-by

$$[\![\text{Sum}_{\vec{A}}\, Q]\!](\mathbf{D},\vec{b}) := \vec{t} \mapsto \begin{cases} \displaystyle\sum_{\{\vec{t}\}\bowtie\{\vec{s}\}=\{\vec{s}\}}^{\mathbb{Q}^2} [\![Q]\!](\mathbf{D},\vec{b})(\vec{s}) \;..\; \text{dom}(\vec{t}) = \vec{A}, \\ \qquad\qquad \{\vec{b}\} \bowtie \{\vec{t}\} \neq \emptyset \\ \langle 0, 0\rangle \qquad\qquad ..\; \text{otherwise} \end{cases}$$

Variable Assignment

$$[\![x := Q]\!](\mathbf{D},\vec{b}) := \vec{t} \mapsto \begin{cases} \langle 1, 1\rangle \;..\; \vec{t}_2 = \langle x_i : t(x_i) \mid x_i \neq x\rangle, \exists m.\ m \neq 0, \\ \qquad \langle m, \vec{t}(x)\rangle = [\![Q]\!](\mathbf{D},\vec{b})(\vec{t}_2) \\ \langle 0, 0\rangle \;..\; \text{otherwise} \end{cases}$$

Figure 2.1 – The formal evaluation semantics of AGCA ($[\![\cdot]\!]$). The operators $+^{\mathbb{Q}^2}$, $*^{\mathbb{Q}^2}$ and $\sum^{\mathbb{Q}^2}$ are vector-wise instances of $+$, $*$ and $\sum$ respectively.

Example 2.2.2 Let R be a GMR of $\mathbb{Q}^2_{\text{Rel}}$

$R^{\mathbf{D}}$	A	B	
	1	2	$\mapsto \langle m_1, q_1\rangle$
	3	5	$\mapsto \langle m_2, q_2\rangle$
	4	2	$\mapsto \langle m_3, q_3\rangle$

where m_i, q_i denote rational multiplicities. Then,

$[\![R(x,y)]\!](\mathbf{D},\langle x:3\rangle)$	x	y	
	3	5	$\mapsto \langle m_2, q_2\rangle$

The query renames the columns (A, B) to (x, y) and selects on x since it is a bound variable.

The AGCA version of the query $\sigma_{A<B}(R)$ evaluates to

$[[R(x,y)*(x<y)]](\mathbf{D},\langle\rangle)$	x	y	
	1	2	$\mapsto \langle m_1, q_1\rangle$
	3	5	$\mapsto \langle m_1, q_2\rangle$

For instance,

$$\begin{aligned}
&[[R(x,y)*(x<y)]](\mathbf{D},\langle\rangle)(\langle x:1,y:2\rangle)\\
&\quad= \sum_{\{\langle x:1,y:2\rangle\}=\{\vec{r}\}\bowtie\{\vec{s}\}}^{\mathbb{Q}^2} [[R(x,y)]](\mathbf{D},\langle\rangle)(\vec{r}) * [[x<y]](\mathbf{D},\vec{r})(\vec{s})\\
&\quad= [[R(x,y)]](\mathbf{D},\langle\rangle)(\langle x:1,y:2\rangle) *^{\mathbb{Q}^2} [[x<y]](\mathbf{D},\langle x:1,y:2\rangle)(\langle\rangle) = \langle m_1,q_1\rangle
\end{aligned}$$ □

In the Sum operator definition from Figure 2.1, for the given tuple $\vec{t}$, we look for all possible tuples $\vec{s}$ that satisfy $\{\vec{t}\}\bowtie\{\vec{s}\}=\{\vec{s}\}$, that is, all possible extensions of tuple $\vec{t}$. There are infinitely many such extensions. When evaluating Q, only those $\vec{s}$ tuples that are present in the database $\mathbf{D}$ will return non-zero multiplicities and all others will evaluate to 0. So, the result of the Sum operator is again an infinite GMR with finite support.

Example 2.2.3 The sum-aggregate query $\mathrm{Sum}_{[y]}(R(x,y)*2*x)$ generalizes the SQL query `SELECT B,SUM(2*A) FROM R GROUP BY B` to GMRs. On the GMR of Example 2.2.2, it yields:

$[[\mathrm{Sum}_{[y]}(R(x,y)*2*x)]](\mathbf{D},\langle\rangle)$	y	
	2	$\mapsto \langle m_1+m_3, 2q_1+8q_3\rangle$
	5	$\mapsto \langle m_2, 6q_2\rangle$

For instance,

$$\begin{aligned}
&[[\mathrm{Sum}_{[y]}(R(x,y)*2*x)]](\mathbf{D},\langle\rangle)(\langle y:2\rangle)\\
&= \sum_{\vec{r},\vec{s},\vec{t}}^{\mathbb{Q}^2} [[R(x,y)]](\mathbf{D},\langle\rangle)(\vec{r}) *^{\mathbb{Q}^2} [[2]](\mathbf{D},\vec{r})(\vec{s}) *^{\mathbb{Q}^2} [[x]](\mathbf{D},\vec{r}\vec{s})(\vec{t})\\
&= \langle m_1*1, q_1*2\rangle *^{\mathbb{Q}^2} [[x]](\mathbf{D},\langle x:1,y:2\rangle)(\langle\rangle) +^{\mathbb{Q}^2} \langle m_3*1, q_3*2\rangle *^{\mathbb{Q}^2} [[x]](\mathbf{D},\langle x:4,y:2\rangle)(\langle\rangle)\\
&= \langle m_1+m_3, 2*q_1+8*q_3\rangle
\end{aligned}$$

The count multiplicity is built-in for each operator to distinguish when the count multiplicity of a tuple is 0 and when the sum aggregate evaluates to 0. □

Using the assignment operator, variables can also take on values of non-grouping aggregates or those that evaluate to a single value for a given set of bindings. That way we can express queries with nested aggregates, which may be correlated with the outside as usual in SQL.

Example 2.2.4 Let relation R and S have columns (A,B) and (C,D). The SQL query

```
SELECT * FROM R
WHERE B < (SELECT SUM(D) FROM S WHERE A > C)
```

is equal to $\mathrm{Sum}_{[A,B]}\big(R(A,B)*(z:=Q_n)*(B<z)\big)$ with $Q_n=\mathrm{Sum}_{[\,]}(S(C,D)*(A>C)*D)$. □

AGCA has no explicit syntax for universal quantification or aggregates other than Sum, but these features can be expressed using (nested) sum-aggregate queries. Special handling of these features in delta processing and query optimization could yield performance better than what we report in our experiments. However, granting these definable features specialized treatment is beyond the scope of this thesis. As a consequence, our implementation provides native support for only the fragment presented above, and the experiments use only techniques described in the thesis. This language specification covers all of the core features of SQL with the exception of null values and outer joins.

2.2.3 Delta Queries

Delta queries express changes in query results for updates to the database. AGCA has the nice property of being *closed under taking deltas*. For any query expression Q, there is an expression ΔQ of *the same language* that captures the change in the result of Q as the database $\mathbf{D}$ is updated by $\Delta\mathbf{D}$, written as

$$\Delta Q(\mathbf{D}, \Delta\mathbf{D}) := Q(\mathbf{D} + \Delta\mathbf{D}) - Q(\mathbf{D}).$$

We define one delta derivation rule for each operator of the query language. Due to the strong compositionality of the language, we can syntactically turn any AGCA expression into its delta by repeatedly applying these rules until we obtain an AGCA expression over GMRs and delta GMRs (updates). We write u to denote an update, and $\Delta_u Q$ for the delta of expression Q with respect to that update. Thus for a GMR R, $\Delta_u R$ is the change to R made in update u.

$$\begin{aligned}
\Delta_u(Q_1 + Q_2) &:= (\Delta_u Q_1) + (\Delta_u Q_2) \\
\Delta_u(Q_1 * Q_2) &:= ((\Delta_u Q_1) * Q_2) + (Q_1 * (\Delta_u Q_2)) \\
&\quad + ((\Delta_u Q_1) * (\Delta_u Q_2)) \\
\Delta_u - Q &:= -\Delta_u Q \\
\Delta_u c &:= 0 \\
\Delta_u x &:= 0 \\
\Delta_u(x \,\theta\, 0) &:= 0 \\
\Delta_u(x := Q) &:= (x := (Q + \Delta_u Q)) - (x := Q) \\
\Delta_u(\mathrm{Sum}_{\vec{A}} Q) &:= \mathrm{Sum}_{\vec{A}}(\Delta_u Q)
\end{aligned}$$

The correctness of the rules follows from the fact that the GMRs with + and $*$ form a ring (for example, the delta rule for $*$ is a direct consequence of distributivity) and that $\mathrm{Sum}_{\vec{A}}$ can be thought of as the repeated application of the + operation [94].

The special case of single-tuple updates is interesting since it allows us to simplify delta queries further and to generate particularly efficient view refresh code. We write $\pm R(\vec{t})$ to denote the insertion/deletion of a tuple $\vec{t}$ into/from relation R of the database.

$$\Delta_{\pm R(\vec{t})}\big(R(x_1,\ldots,x_{|\mathrm{sch}(R)|})\big) := \pm \prod_{i=1}^{|\mathrm{sch}(R)|} (x_i := t_i)$$
$$\Delta_{\pm R(\vec{t})}\big(S(x_1,\ldots,x_{|\mathrm{sch}(S)|})\big) := 0 \qquad (R \neq S)$$

Example 2.2.5 Consider the AGCA query computing a sum aggregate:

$$Q = \mathrm{Sum}_{[\,]}\big(R(A,B) * S(C,D) * (B = C) * A * D\big)$$

We abbreviate the Sum subexpression as α. Let us study the insertion/deletion of a single tuple $\langle A : x, B : y\rangle$ to/from R. From the delta rule for R, $\Delta_{\pm R(x,y)} R(A,B) = \pm(A := x) * (B := y)$, and from the delta rule for $*$:

$$\begin{aligned}\Delta_{\pm R(x,y)}\alpha &= \pm(A := x) * (B := y) * S(C,D) * (B = C) * A * D \\ &= \pm S(C,D) * (y = C) * x * D\end{aligned}$$

The delta of the main query is $\Delta_{\pm R(x,y)} Q = \mathrm{Sum}_{[\,]}(\Delta_{\pm R(x,y)}\alpha)$. □

Example 2.2.6 Consider the overall query with a nested aggregate from Example 2.2.4. The delta for insertion/deletion of a tuple $\langle C : x, D : y\rangle$ to/from relation S is:

$$\Delta_{\pm S(x,y)} Q = \mathrm{Sum}_{[\mathrm{A,B}]}\big(R(A,B) * \Delta_{\pm S(x,y)}(z := Q_n) * (B < z)\big)$$

Following the delta rule for $:=$,

$$\Delta_{\pm S(x,y)}(z := Q_n) = \big(z := (Q_n \pm \Delta_{\pm S(x,y)} Q_n)\big) - (z := Q_n)$$

where $\Delta_{\pm S(x,y)} Q_n = \mathrm{Sum}_{[\,]}\big((C := x) * (D := y) * (A > C) * D\big) = (A > x) * y$, which is a way of writing "if $(A > x)$ then y else 0". □

2.2.4 Binding Patterns

AGCA query expressions have input and output variables and incorporate binding patterns representing information flow. *Input variables* are parameters whose values cannot be computed from the database and must be provided from the outside to evaluate the query. *Output variables* represent columns of the query result schema.

The most interesting case of input variables occurs in a correlated nested subquery, viewed in isolation. In such a subquery, a correlation variable from the outside is such an input variable. The subquery can only be computed if a value for the input variable is given.

Example 2.2.7 In Example 2.2.4, all columns of R's schema are output variables. In the subexpression Q_n, A is an input variable and there are no output variables since the aggregate is non-grouping. Taking a delta *adds input variables*, parameterizing the query with the update. In Example 2.2.5, the delta query uses input variables x and y to pass the update. In Example 2.2.6, the delta query $\Delta_{\pm S(x,y)} Q_n = (A > x) * y$ has input variables A, x, and y. □

2.3 The Viewlet Transform

We now describe the viewlet transform. In this section, we focus only on flat queries and exclude variable assignments $x := q$ where q contains a sum-aggregate from the query language. We eliminate this restriction in the next chapter. This query language fragment has the following nice property. ΔQ is structurally strictly simpler than Q when query complexity is measured using the *degree* of query Q, denoted by $\deg(Q)$. For union-free queries, $\deg(Q)$ is the number of relations joined together. For queries with unions, $\deg(Q)$ is the maximum degree of the union-free subqueries obtained after pushing unions above joins based on the distributivity of addition over multiplication. Queries are strongly analogous to polynomials, and the degree of queries is defined precisely as it is defined for polynomials (where the relation atoms of the query correspond to the variables of the polynomial).

Theorem 2.3.1 ([94]) If $\deg(Q) > 0$ then $\deg(\Delta Q) = \deg(Q) - 1$.

The viewlet transform uses the simple fact that a delta query is a query too. Thus it can be incrementally maintained using the delta query of the delta query, which again can be materialized and incrementally maintained, and so on, recursively. By the above theorem, this recursive query transformation terminates at the $\deg(Q)$-th recursion level, when the obtained delta is a "constant" (degree 0) independent of the database and dependent only on updates.

In the following, we write $\Delta^l Q[u_1,\ldots,u_l]$ $(l \geq 0)$ to denote a view representing the query $\Delta_{u_l} \cdots \Delta_{u_1} Q$ parameterized by updates $u_1,\ldots,u_l$. In general, this is a higher-order delta query, but the case $l = 0$ is simply the query Q. We consider an update to be a database of GMRs that potentially holds inserts and deletes for any base relation of the database being updated.

Definition 2.3.2 Given a query Q, the viewlet transform turns Q into the following update trigger, which maintains the view of Q plus a set of auxiliary views parametrized by updates u_1, u_2, and so on. When maintaining these auxiliary views, update u binds one of these parameters at runtime while for the others we need to loop over all different valuations of $u_1,\ldots u_k$ that we have seen so far.

Listing 2.3 The pseudocode of the viewlet transform

on update u **do**:
  for $k = 0$ **to** $\deg(Q) - 1$ **do**
    foreach $u_1,\ldots,u_k \in$ {domain of RHS expr given u} **do**

$$\Delta^k Q[u_1,\ldots,u_k] \mathrel{+}= \begin{cases} \Delta_u \Delta_{u_k} \cdots \Delta_{u_1} Q & \ldots \text{ if } k = \deg(Q) - 1 \\ \Delta^{k+1} Q[u_1,\ldots,u_k,u] & \ldots \text{ otherwise} \end{cases}$$

□

The viewlet transform owes its name to a superficial analogy with the Haar wavelet transform, which also materializes a hierarchy of differences.

The above example shows that each trigger statement loops over the domains of the variables $u_1,\ldots,u_k$ at runtime. When these domains are large, such iterations might become

prohibitively expensive. One way to make the viewlet transform practical is to restrict the updates to be constant-size batches. Without true loss of generality, we focus on single-tuple updates as they offer particular optimization potential. But, they also require multiple triggers to handle different update events, namely insertions and deletions for multiple relations.

We create insert and delete trigger functions in which the argument is a tuple (i.e., a list of tuple variables) rather than a GMR. Using primitive-typed variables avoids looping in trigger statements and enables more powerful query rewrites and simplifications, as shown in the following example. We discuss these optimizations in detail in the next chapter.

Example 2.3.3 Consider the second-degree query Q from Example 2.2.5 with single-tuple updates. We write $\pm R(x, y)$ to denote the insertion/deletion of a tuple $\langle x, y\rangle$ into/from relation R. Let $\text{sgn}_R, \text{sgn}_S \in \{+, -\}$. Then, one of the second-order deltas is

$$(\Delta_{\text{sgn}_R R(x,y)} \Delta_{\text{sgn}_S S(z,u)} Q)[x, y, z, u] = \text{sgn}_R \text{sgn}_S \, (y = z) * x * u$$

A trigger for events $\pm R(x, y)$ can be obtained as follows. Variables x and y are arguments of the trigger and are bound at runtime, but variables z and u need to be looped over. On the other hand, the right-hand side of the trigger is only non-zero in case that $y = z$. So we can substitute z by y everywhere and eliminate z. Using this simplification, the viewlet transform produces the following trigger for $+R(x, y)$:

Listing 2.4 Trigger for single-tuple updates to R

Q += $(\Delta_{+R(x,y)} Q)[x, y]$;
foreach u **do** $\Delta_{+S(y,u)} Q[y, u]$ += $x * u$;
foreach u **do** $\Delta_{-S(y,u)} Q[y, u]$ −= $x * u$;

The construction of the remaining triggers happens analogously. The trigger contains an update rule for the (in this case, scalar) view Q for the overall query result. The rule uses the auxiliary view $\Delta_{\pm R(x,y)} Q$, which is maintained in the update triggers for S. The trigger also contains update rules for the auxiliary views $\Delta_{\pm S(y,u)} Q$ that are used to update Q in the update triggers for S. The reason why we omitted deltas $\Delta_{\pm R(\ldots)} \Delta_{\pm R(\ldots)} Q$ and $\Delta_{\pm S(\ldots)} \Delta_{\pm S(\ldots)} Q$ is because these are guaranteed to be 0 as the query has no self-join. An additional optimization, presented in the next chapter, can eliminate the loops on u using the distributivity and associativity of the + and $*$ operations. □

We observe that the structure of the work that needs to be done is extremely regular and (conceptually) simple. Moreover, there are no classical coarse-grained query operators left, so it makes no sense to give this workload to a classical query optimizer. There are for-loops over many variables, which have the potential to be very expensive. But the work is also perfectly data-parallel, and there are no data dependencies comparable to those present in joins. All this provides justification for making heavy use of compilation.

2.4 Summary

In this chapter, we explain the basic idea of incremental view maintenance in databases. We introduce the data model and the query language that we will use in the following sections, and we define a set of rules for computing delta queries. Then, we present a recursive query compilation algorithm, called the viewlet transform, which materializes a set of higher-order delta views with the goal of lowering view maintenance cost.

We refer to the viewlet transform as presented in this chapter as the naïve viewlet transform. Next, we present improvements and optimizations that make the viewlet transform practical.

3 Higher-Order Incremental View Maintenance

This chapter presents a practical implementation of the viewlet transform called *Higher-Order Incremental View Maintenance*. Like the viewlet transform, Higher-Order IVM transforms a query Q into a *trigger program* — a set of triggers that maintain the materialized view (as a *map* or *dictionary*) M_Q on query Q, and a set of supplemental materialized views. As before, each trigger consists of *update statements*, each of the form **foreach** $\vec{x}$ **do** $M_Q[\vec{x}]$ += $Q'[\vec{x}]$.

The naïve viewlet transform may produce delta queries that are very expensive or simply impossible to maintain. An example of the former is a delta query including a Cartesian product (i.e., a product of two subqueries $Q_1 * Q_2$ with no output variables in common). As we will soon see, such queries arise quite frequently in the viewlet transform and have high maintenance costs. Delta queries that are impossible to maintain include (1) deltas that contain input variables and therefore lack finite support, and (2) deltas of queries with nested subqueries, to which Theorem 2.3.1 does not apply.

The key insight behind Higher-Order IVM is that full materialization of entire delta queries is unnecessary. When generating update statements for a materialized view Q, we materialize the delta terms $\Delta_u Q$ as one or more subqueries of each delta query. These subexpressions are then combined together to compute $\Delta_u Q$ when executing the corresponding update statements. Materializing the delta query piecewise increases the *execution cost* of evaluating trigger statements. However, by carefully selecting an appropriate set of subqueries, the increased evaluation overhead is offset by a substantial reduction in *view maintenance costs*.

We now formally define Higher-Order IVM (HO-IVM). Recall that the viewlet transform produces a sequence of statements, each of the form: $Q[\vec{x}]$ += $\Delta_u Q[\vec{x}]$. Unlike the viewlet transform which materializes $\Delta_u Q$ as a single view, HO-IVM materializes a set of subqueries $\vec{M}_{\Delta_u Q}$ and rewrites the statement into an equivalent statement $Q[\vec{x}]$ += $\Delta_u Q'[\vec{x}]$, evaluated over these materialized views. Then, instead of recurring on $\Delta_u Q$ as in the viewlet transform, HO-IVM recurs individually on each $M_i \in \vec{M}_{\Delta_u Q}$. We refer to the rewritten query and the set of materialized subqueries as a *materialization decision* for $\Delta_u Q$, denoted by $\langle \Delta_u Q', \vec{M}_{\Delta_u Q} \rangle$.

Example 3.0.1 Consider the following query:

$$Q[\,] = \mathrm{Sum}_{[\,]}(R(A,B) * S(B,C) * T(C,D))$$

The insertion trigger for $+S(b,c)$ includes the statement $Q[\,]$ += $\mathrm{Sum}_{[\,]}(R(A,b) * T(c,D))$. The naïve viewlet transform materializes the entire expression $\mathrm{Sum}_{[b,c]}(R(A,b) * T(c,D))$, whereas HO-IVM expresses Q in terms of two sub-expressions, $M_1[b] := \mathrm{Sum}_{[b]}(R(A,b))$ and $M_2[c] := \mathrm{Sum}_{[c]}(T(c,D))$, which are maintained separately. The insertion trigger then includes the statement $Q[\,]$ += $\mathrm{Sum}_{[\,]}(M_1[b] * M_2[c])$. □

Algorithm 3.1 summarizes HO-IVM and Sections 3.1–3.2 discuss heuristics for obtaining a materialization decision (which define the `materialize()` procedure).

Algorithm 3.1 HO-IVM(Q, M_Q) – Higher-Order IVM

Require: A query Q to be maintained as M_Q
Ensure: A list of update statements T_u for each update event u
for all Relation Name R used in Q **do**
  for all $u \in \{+R, -R\}$ **do**
    let $\vec{x}$ = the input/output variables of $\Delta_u Q$
    let $\langle Q', \{M_i := Q_i\}\rangle$ = `materialize`$(\Delta_u Q)$
    update $T_u = T_u$ :: (**foreach** $\vec{x}$ **do** $M_Q[\vec{x}]$ += $Q'[\vec{x}]$)
    for all i **do** `HO-IVM`(Q_i, M_i)
  end for
end for

Note that we can partially materialize not only delta queries but also user-provided (top-level) queries. Although this strategy introduces a computational overhead on every view access, it can substantially reduce view maintenance costs in certain cases. For instance, we can benefit from piecewise materialization when computing averages as we need to maintain two separate, simpler aggregates: the count and the sum. Reconstructing the average value from these partial aggregates is a constant time operation which can be done on-the-fly with every view access. This generalized form of Higher-Order IVM is made explicit in Algorithm 3.2.

Algorithm 3.2 Generalized Higher-Order IVM(Q)

Require: A query Q to be maintained
Ensure: A query Q', equivalent to Q
Ensure: A list of update statements T_u for each update event u
  let $\langle Q', \{M_i := Q_i\}\rangle$ = `materialize`(Q)
  for all i **do** `HO-IVM`(Q_i, M_i)

Query Decomposition

$$\mathcal{M}(\text{Sum}_{\vec{A}\vec{B}}(Q_1 * Q_2)) \;\Rightarrow\; \mathcal{M}(\text{Sum}_{\vec{A}}(Q_1)) * \mathcal{M}(\text{Sum}_{\vec{B}}(Q_2)) \quad (1)$$

$\vec{A}$ and $\vec{B}$ are any disjoint sets of variables.

Factorization and Polynomial Expansion

$$\mathcal{M}(\text{Sum}_{\vec{A}}(Q_L * (Q_1 + Q_2 + \ldots) * Q_R)) \;\Leftrightarrow$$
$$\mathcal{M}(\text{Sum}_{\vec{A}}(Q_L * Q_1 * Q_R)) + \mathcal{M}(\text{Sum}_{\vec{A}}(Q_L * Q_2 * Q_R)) + \ldots \quad (2)$$

Input Variables

$$\mathcal{M}(\text{Sum}_{\vec{A}}(Q * f(\vec{B}\vec{C}))) \;\Rightarrow\; \text{Sum}_{\vec{A}}(\mathcal{M}(\text{Sum}_{\vec{A}\vec{B}}\, Q) * f(\vec{B}\vec{C})) \quad (3)$$

Q is the maximal subquery that contains no input variables.
f is a subquery that contains no relation terms
$\vec{A}$ is any set of variables and $\vec{B}$ is the set of output variables of Q referenced by f
$\vec{C}$ is the set of input variables referenced by f

Nested Aggregates and Decorrelation

$$\mathcal{M}(\text{Sum}_{\vec{A}}(Q_O * (x := Q_N) * f(x, \vec{B}))) \;\Rightarrow\; \text{Sum}_{\vec{A}}(\mathcal{M}(\text{Sum}_{\vec{A}\vec{B}}(Q_O)) * (x := \mathcal{M}(Q_N)) * f(x, \vec{B})) \quad (4)$$

Q_O is the maximal subquery for which x is not an *input* variable.
Q_N is a subquery containing at least one relation term.
f is a subquery containing no relation terms.
$\vec{A}$ is any set of variables and $\vec{B}$ is the set of output variables of Q_O referenced by f or Q_N

Figure 3.1 – Rewrite rules for partial materialization. Bidirectional arrows indicate rules applied heuristically from left to right during materialization but also in reverse to some expressions. Note that for any query Q with output variables $\vec{A}$, the property $Q = \text{Sum}_{\vec{A}}(Q)$ holds.

3.1 Heuristic Optimization

In this section, we present a set of heuristic rewrite rules for partial materialization of a given query. We repeatedly apply these rules starting from the naïve materialization decision $\langle (M_{Q,1}), \{M_{Q,1} := Q\}\rangle$ and until reaching a fixed point. For clarity, we present these rules in terms of a materialization operator $\mathcal{M}$. For example, one possible materialization decision for $Q := Q_1 * Q_2$ is $\mathcal{M}(Q_1) * \mathcal{M}(Q_2) \equiv \langle (M_{Q,1} * M_{Q,2}), \{M_{Q,i} := Q_i\}\rangle$.

Figure 3.1 presents all but the trivial rewrite rules for partial materialization. We discuss the full array of heuristic optimizations in depth below. Figure 3.2 shows how these rules apply to the experimental workload discussed in Section 4.4.

3.1.1 Duplicate View Elimination

The viewlet transform produces many duplicate views, mainly because the delta operation typically commutes with itself. For instance, $\Delta_R \Delta_S Q = \Delta_S \Delta_R Q$ for any Q without nested aggregates over R or S. Structural equivalence on the view definition queries is typically sufficient to identify this type of view duplication. View deduplication, as the simplest optimization, substantially reduces the number of views created.

	Query	Features					Rules			
		#Tables	Join	Predicate	GroupBy	Nesting	Decomp	PolyExp	InputVar S/C	Nested R/I
TPC-H	Q1	1		<	✓	-	-	✓	S	I
	Q2	5	=	∧,=	-	1	✓	✓	S	I
	Q3	3	=	∧,<	✓	-	✓	✓	-	-
	Q4	1		∧,<	✓	1	-	✓	S	I
	Q5	6	=	∧,<	✓	-	✓	✓	-	-
	Q6	1		∧,<	-	-	-	-	-	-
	Q7	6	=	∧,∨,<	✓	1	✓	✓	-	-
	Q8	7	=	∧,=,<	✓	1	✓	✓	S	R
	Q9	6	=	∧,=	✓	1	✓	✓	-	-
	Q10	4	=	∧,=,<	✓	-	✓	✓	-	-
	Q11	2	=	-	✓	-	✓	✓	S	I
	Q12	2	=	∧,=,<	✓	-	-	✓	-	-
	Q13	2	=	≠	✓	1	-	✓	S	I
	Q14	2	=	∧,<	-	-	-	✓	S	R
	Q15	2	=	∧,<	✓	2	-	✓	S	I
	Q16	2	=	∨,=,≠	✓	1	✓	✓	S	R
	Q17	2	=	<	-	1	✓	✓	S	I
	Q18	3	=	<	✓	2	✓	-	S	R,I
	Q19	2	=	∨,=,<	-	-	-	✓	-	-
	Q20	2	=	∧,=,<	-	2	-	✓	S	I
	Q21	4	=	∧,=,<	✓	1	✓	✓	S	I
	Q22	1		=,<	✓	1	-	✓	S	R,I
Finance	AXF	2	=	∨,<	✓	-	-	✓	S	-
	BSP	2	=	∧,<	✓	-	-	✓	-	-
	BSV	2	=	-	-	-	-	✓	-	-
	MST	2	x	∧,<	✓	1	-	✓	S	R,I
	PSP	2	x	∧,<	-	1	-	✓	S	R,I
	VWAP	1		<	-	1	-	✓	C	R

Figure 3.2 – Workload features and rewrite rules applied to each query. *Features notation:* Number of join tables, Join type (=: equi, x: cross), Predicate type (∧: conjunction, ∨: disjunction, =: equality, ≠: inequality, <: range inequality), GroupBy clause, Nesting depth. *Rules notation:* Query decomposition, Factorization and polynomial expansion, Input variables with a subquery (S) or a view cache (C) (see §3.6.3), Nested aggregates and decorrelation with re-evaluation of the nested query (R) or incremental evaluation (I).

3.1.2 Query Decomposition

Queries with disconnected join graphs are particularly expensive to materialize. If the join graph of Q includes multiple disconnected components Q_1, Q_2, ... (i.e., Q is the Cartesian product $Q_1 \times Q_2 \times \ldots$), it is better to materialize each component independently as $\mathcal{M}(Q_1) * \mathcal{M}(Q_2) * \ldots$ instead of a single view $\mathcal{M}(Q)$. The cost of selecting from (iterating over) $\mathcal{M}(Q)$ is similar to the cost of selecting from $\mathcal{M}(Q_1) * \mathcal{M}(Q_2) * \ldots$, as both require an iteration over $|Q_1| \times |Q_2| \times \ldots$ elements. Furthermore, maintaining each individual Q_i is less computationally expensive: the decomposed materialization stores (and maintains) only $|Q_1| + |Q_2| + \ldots$ values, while the combined materialization handles $|Q_1| * |Q_2| * \ldots$ values.

The query decomposition rewrite rule presented in Figure 3.1.1 exploits the generalized distributive law [27] to break up such queries with Sum aggregates into smaller components for materialization. These cases often arise in the viewlet transform. For instance, taking a delta of a query with respect to a single-tuple update replaces a relation in the query by a singleton constant tuple, effectively eliminating one hyperedge from the join graph and creating new disconnected components that can be further decomposed. Consequently, this optimization plays a major role in the efficiency of our implementation (see Example 3.0.1), and for ensuring that the number of maps created for any acyclic query is polynomial.

3.1.3 Polynomial Expansion and Factorization

The described query decomposition operates exclusively over conjunctive queries (i.e., AGCA expressions without addition). To support decomposition across unions, we observe that addition and aggregate summations commute, $\text{Sum}_{\vec{A}}(Q_1 + Q_2) = \text{Sum}_{\vec{A}}(Q_1) + \text{Sum}_{\vec{A}}(Q_2)$, and that the generalized distributive law [27] applies, $Q_1 * (Q_2 + Q_3) = (Q_1 * Q_2) + (Q_1 * Q_3)$. Consequently, any query can be expanded into a sum of multiplicative clauses, where each clause is a conjunctive query (analogous to a query in disjunctive normal form).

Our heuristic-based materialization strategy fully expands queries into normal form, the process we refer to as polynomial expansion, in order to materialize each multiplicative clause independently and enable an effective use of query decomposition. Figure 3.1.2 shows the rewrite rule for polynomial expansion.

Note that this rule is bidirectional. If a common term (Q_L and Q_R in the rewrite rule) appears in several multiplicative clauses, the term can be *factored* out of the sum of these multiplicative clauses for an equivalent, smaller, and cheaper query expression. It is often possible (and beneficial) to factorize the rewritten query Q' after obtaining a final materialization decision $\langle Q', \{\ldots\}\rangle$, when the expression is no longer required to be in normal form.

3.1.4 Input Variables

The delta operation introduces input variables, which in turn makes it possible to create delta queries without finite support. For example, consider the query:

$$Q[A,B,C] = R(A,B) * S(C) * (B < C) * A$$

The delta query $\Delta_{+R(x,y)} Q[x,y,C] = S(C) * (y < C) * x$ has two input variables (x,y), making it impossible to fully materialize it.

A trivial solution to this problem is to simply avoid materializing terms that contain input variables, which is precisely the aim of the rewrite rule in Figure 3.1.3. The rule exploits the generalized distributive law [27] to pull terms containing input variables out of the materialization operator. As with query decomposition, this rewrite rule assumes the multiplication operation at the root of the query expression tree.

In addition to extracting input variables from the materialized query, this rewrite rule also pushes summation into the materialized expression[1]. This is analogous to a common optimization in query processing where aggregation and projection operators are pushed down as far as possible into the evaluation pipeline.

In addition to the trivial solution, several other strategies for dealing with input variables are possible. For queries where the input variables appear in comparison predicates (e.g., $S(C) * (y < C)$), data structure-based solutions like range indices are possible but beyond the scope of this work. A third strategy based on caching is discussed below.

3.1.5 Deltas of Nested Aggregates

AGCA encodes nested subqueries using the assignment operator ($:=$). Recall that the delta rule for this operator is:

$$\Delta_u(x := Q) := (x := Q + \Delta_u Q) - (x := Q)$$

The delta query references the original query (twice), and is clearly not simpler than the original query (as per Theorem 2.3.1). On such expressions, the (naïve) viewlet transform fails to terminate. Of course, queries with assignment are not always catastrophic. If $\Delta_u Q = 0$, then

$$(x := Q + \Delta_u Q) - (x := Q) = (x := Q) - (x := Q) = 0$$

For assignments where the query Q being assigned to x corresponds to a simple arithmetic expression, the delta is always empty. However, if Q contains a relation term $R(\vec{A})$ (i.e., Q represents a nested subquery), then the deltas $\Delta_{\pm R}$ must be handled as a special case.

The rewrite rule from Figure 3.1.4 uses the generalized distributive law [27] to identify nested subqueries that need to be materialized. Piecewise materialization enforced by the rule is key for Higher-order IVM to terminate when compiling queries with nested aggregates. As with rules 3.1.1 and 3.1.3, this rule relies on polynomial expansion to simplify the expression into a sum of multiplicative clauses, and like rule 3.1.3, it aggressively pushes aggregates down into the newly created expressions.

Although this rule is necessary to guarantee compiler termination, it can introduce unnecessary overheads into the evaluation of delta queries. When naïvely used, this rule might separately materialize a nested subquery that does not reference the delta relation. A refinement of this optimization analyzes a given delta query before applying the rewrite rule: Considering the expression from Figure 3.1.4 and update u to relation R, it is only necessary to apply the rewrite rule to Q_N when Q_N includes a reference to R. If it does not, then $\Delta_u Q_N = 0$, and the rewrite is not needed to ensure the termination of Higher-Order IVM.

[1] We might omit the sum aggregation in some cases because an expression Q with output variables $\vec{A}$ is equivalent to the expression $\text{Sum}_{\vec{A}}(Q)$.

Example 3.1.1 Recall the delta query $\Delta_{\pm S(x,y)} Q$ from Example 2.2.6.

$$\Delta_{\pm S(x,y)} Q \;=\; \mathrm{Sum}_{[\mathrm{A,B}]}(R(A,B) * (z := (Q_n \pm (A > x) * y)) * (B < z) - R(A,B) * (z := Q_n) * (B < z))$$

where $Q_n = \mathrm{Sum}_{[\,]}(S(C,D) * (A > C) * D)$. The materialization decision for this delta query materializes two subqueries, $M_{Q,1} := R(A,B)$ and $M_{Q,2} := \mathrm{Sum}_{[\mathrm{C}]}(S(C,D) * D)$. On every update of relation S, the delta evaluation effectively evaluates the outer query twice: once using the new value $Q_n \pm \Delta Q_n$ and once using the old value of Q_n. Conversely, the delta for updates to R, $\Delta_{\pm R} Q$ always has a lower degree than Q, and is materialized as a single map (the nested-aggregates rewrite rule is ignored). □

This example reveals one additional possibility for optimizing nested aggregates. The delta for insertions into S is actually *more* expensive than re-evaluating the entire update (The outer query is evaluated twice in the delta, but just once in the original). Thus, in some situations we might want produce an update statement that replaces the map being maintained, instead of updating it. As a general rule, the incremental approach pays off when the inner query is correlated on an equality, and the delta's arguments bind at least one of these variables; then the delta query only aggregates over a subset of the tuples in the outer query. For instance, if the nested query from Example 2.2.6 were to have $(A = C)$ instead of $(A > C)$, then only a subset of the aggregated tuples would have been affected by the delta, suggesting the incremental approach is a better choice. Based on this analysis, the heuristic optimizer decides whether to re-evaluate or incrementally maintain any given delta query.

Note that although Q is being recomputed, we can still accelerate the computation by materializing Q piecewise. Although the expression being materialized is not a delta, we still compute a materialization decision (as in Generalized Higher-Order IVM).

Because we are already materializing the expression Q, care must be taken to avoid creating a self-referential loop in this materialization decision. The default materialization decision $\mathcal{M}(Q)$ is meaningless, as Q defines the view being maintained. We avoid this by first applying the nested-query rewrite heuristic as aggressively as possible to eliminate all nested subqueries in the expressions being materialized. Because recomputation is only appropriate for queries with nested subqueries (otherwise it is better to perform IVM), the resulting expression is guaranteed to be simpler than Q.

3.2 Simplifying Delta Expressions

Although the delta operation reduces the expression degree, it introduces input variables and tends to make the expression itself longer and more complicated. For products and some conditions, it creates additional additive terms.

Example 3.2.1 Consider the following expression:

$$Q[A,B] = R(A) * R(A) * S(B)$$

Applying the delta rules leaves us with the expression:

$$\begin{aligned}\Delta_{+R(x)} Q[A,B] = \big((A:=x) * R(A) + R(A) * (A:=x) + (A:=x) * (A:=x)\big) * S(B) + \\ R(A) * R(A) * 0 + \big((A:=x) * R(A) + R(A) * (A:=x) + (A:=x) * (A:=x)\big) * 0\end{aligned}$$

This expression is complex but can be simplified to $\Delta_{+R(x)} Q[x,B] = (2 * R(x) + 1) * S(B)$. □

This added complexity increases both compilation and evaluation costs. As part of Higher-Order IVM, we regularly apply several simplifying transformations to AGCA expressions, some of which correspond to common relational algebra transformations. Like with the heuristic rules, these simplifications are applied repeatedly, up to a fixed point.

3.2.1 Unification

Delta derivation with single-tuple updates creates many variable assignments with constant trigger arguments. We use *unification* to propagate these range restrictions and simplify AGCA expressions. The procedure consists of two steps. First, we transform equality predicates into equivalent assignment expressions, and then we propagate assignments through the expression, eliminating them if appropriate.

In the first stage, we identify equality comparisons that can be rewritten into an assignment-compatible form where a single variable appears on the left-hand side of the term. Each such equality comparison is commuted left through product terms and out of Sum operators until either (1) the left-hand variable falls out of scope, or (2) commuting it further would cause a variable appearing on the right-hand side to fall out of scope. If condition 1 is satisfied, we convert the equality into an assignment.

An equality comparison may have multiple assignment-compatible forms. At most one of these forms can possibly satisfy condition 1 as a second variable falling out of scope would violate condition 2. When multiple forms are available, we continue commuting until condition 1 is satisfied for precisely one form or until condition 2 is violated for all forms.

Once all equality comparisons have been converted into assignments, we propagate assignments throughout the expression. This is analogous to beta reduction in lambda calculus, although there are semantic restrictions on AGCA expressions that can prevent us from fully reducing the assignment. There are three such limitations: (1) AGCA forbids range restrictions to be incorporated directly into relation terms, (2) AGCA disallows computationally intensive (i.e. nested aggregate) expressions to be incorporated directly into comparison operations, and (3) If the assignment creates a range restriction on the domain of the query output, the assignment must remain in the expression.

The assignment is propagated as aggressively as possible. If none of the above limitations are violated and the variable is not in the schema of the query, then the variable is no longer used and the assignment can be safely removed.

3.2.2 Partial Evaluation and Algebraic Identities

Delta derivation frequently produces expressions containing sums of terms that differ only by a constant multiplier. We apply the polynomial factorization heuristic presented in Section 3.1 (ignoring the materialization operator) to group and and sum up the constant multipliers.

During this optimization stage, AGCA expressions are partially evaluated by merging constant values that appear in a sum or product together, and by applying the standard algebraic identities $Q+0=Q$, $Q*1=Q$, and $Q*0=0$. This last identity is especially useful during delta computation, as the delta operation produces many zeroes and expressions of the form $Q-Q$.

3.2.3 Extracting Range Restrictions

Variable assignments that create a range restriction on the output of a query can sometimes be pulled out of the query. The primary application of this technique is for update trigger statements, where a range restriction on the statement's loop variables can be applied directly to the map being updated.

The range restriction procedure is as follows. After the query has been fully simplified, we identify all assignments where the right-hand value is a single trigger variable that can be commuted up to the left-most position of a product term at the root of the query. We extract these assignments from the query to create a mapping from loop variables to trigger variables, which is applied to both the query and the variables of the map being updated.

For instance, consider the delta query from Example 3.2.1. Its simplified version contains terms of the form: $(A:=x)*R(A)*S(B)$. Here, we can extract the assignment and eliminate the loop over variable A in the update statement; the optimized statement is **foreach** B **do** $Q[x,B]$ += $\Delta_R Q[x,B]$. Note that one of the variables appearing on the left-hand side of the update statement has been bound to a corresponding trigger variable (x).

A similar technique is crucial for efficiently maintaining nested aggregate deltas, as seen in the following example.

Example 3.2.2 Consider the query $Q[A,B]=(B:=R(A))$. Let R contains 2 tuples as follows, then $Q[A,B]$ is:

$[[R(A)]](\mathbf{D},\langle\rangle)$	A
	$1 \mapsto \langle 1,1\rangle$
	$2 \mapsto \langle 3,3\rangle$

$[[Q]](\mathbf{D},\langle\rangle)$	A B
	1 $1 \mapsto \langle 1,1\rangle$
	2 $3 \mapsto \langle 1,1\rangle$

The delta of Q for the update $+R(a)$ is: $\Delta_{+R(a)}Q[A,B]=(B:=R(A)+(A:=a))-(B:=R(A))$.

The GMR for the delta with respect to the insertion of $\langle A:1\rangle$ into R includes two tuples with nonzero multiplicities:

$[[\Delta_{+R(1)}Q]](\mathbf{D},\langle\rangle)$	A	B	
	1	1	$\mapsto \langle -1,-1\rangle$
	1	2	$\mapsto \langle 1,1\rangle$

But, while evaluating the delta, two intermediate GMRs are instantiated with $|R|$ tuples each:

$$\left(\begin{array}{|cc|}\hline A & B & \\ \hline 1 & 2 & \mapsto \langle 1,1\rangle \\ 2 & 3 & \mapsto \langle 1,1\rangle \\ \hline\end{array}\right) \quad - \quad \left(\begin{array}{|cc|}\hline A & B & \\ \hline 1 & 1 & \mapsto \langle 1,1\rangle \\ 2 & 3 & \mapsto \langle 1,1\rangle \\ \hline\end{array}\right)$$

Tuples in these GMRs corresponding to tuples with zero multiplicities in $\Delta_{+R(1)}R(A)$ (i.e. $\langle A:2,B:3\rangle$) cancel out. A simpler, equivalent query would be:

$$\Delta_{+R(x)}Q[A,B] = (A:=x) * ((B:=R(A)+1) - (B:=R(A))) \qquad \square$$

Recall the delta rule for nested queries: $\Delta_u(x:=Q) = (x:=Q+\Delta_u Q) - (x:=Q)$. After computing the nested delta $\Delta_u Q = (\Delta_u Q)_{rr} * (\Delta_u Q)_e$, we extract all range restrictions $(\Delta_u Q)_{rr}$ and prepend them to the delta of the full expression. The revised delta rule is:

$$\Delta_u(x:=Q) = (\Delta_u Q)_{rr} * ((x:=Q+(\Delta_u Q)_e) - (x:=Q))$$

Section 3.4 generalizes the range restriction procedure for both single-tuple and batch updates.

3.3 Examples: Putting It All Together

In this section, we provide several examples of Higher-Order IVM. Our goal is to illustrate how the heuristic optimizations interact to produce an efficient view maintenance program and highlight some interesting behaviors.

The examples are rather involved. To promote clarity, we explicitly give output variables with maps. We write $Q[\vec{x_{out}}]$ to denote a map Q with output variables $\vec{x_{out}}$, which form the schema of the query result.

3.3.1 Simplified TPC-H Query 18

We explain the compilation process on a query with an equality-correlated nested aggregate:

```
SELECT C.CK, SUM(LI.QTY) FROM C, O, LI
WHERE C.CK = O.CK AND O.OK = LI.OK AND
      100 < (SELECT SUM(LI1.QTY) FROM LI AS LI1
             WHERE LI.OK = LI1.OK)
GROUP BY C.CK
```

The query is a simplified version of Q18 from our test workload (see Appendix A.1). For simplicity, we use the condensed schema $C(CK)$, $O(CK,OK)$, and $LI(OK,QTY)$. Figure 3.3

Listing 3.3 Single-tuple triggers for the simplified TPC-H Q18

```
1  Top-level and 5 auxiliary materialized views:
2    Q[CK] := Sum_[CK](C(CK)*O(CK,OK)*LI(OK,QTY)*QTY*(x:=Q_n)*(100<x))
3    Q_C[CK] := Sum_[CK](O(CK,OK)*LI(OK,QTY)*QTY*(x:=Q_n)*(100<x))
4    Q_O1[CK] := Sum_[CK](C(CK))
5    Q_O2[OK] := Sum_[OK](LI(OK,QTY)*QTY)
6    Q_LI[CK,OK] := Sum_[CK,OK](C(CK)*O(CK,OK))
7    Q_LI,C[CK,OK] := Sum_[CK,OK](O(CK,OK))
8
9  on insert into C values (ck):
10   Q[ck] += Q_C[ck]
11   foreach OK do Q_LI[ck,OK] += Q_LI,C[ck,OK]
12   Q_O1[ck] += 1
13
14 on insert into O values (ck,ok):
15   Q[ck] += Q_O1[ck]*Q_O2[ok]*(x := Q_O2[ok])*(100 < x)
16   Q_LI[ck,ok] += Q_O1[ck]
17   Q_LI,C[ck,ok] += 1
18   Q_C[ck] += Q_O2[ok]*(x := Q_O2[ok])*(100 < x)
19
20 on insert into LI values (ok,qty):
21   foreach CK do
22     Q[CK] += Q_LI[CK,ok]*(((Q_O2[ok]+qty)*(x := Q_O2[ok]+qty))
23                 - (Q_O2[ok]*(x := Q_O2[ok])))*(100 < x)
24   foreach CK do
25     Q_C[CK] += Q_LI,C[CK,ok]*(((Q_O2[ok]+qty)*(x := Q_O2[ok]+qty))
26                 - (Q_O2[ok]*(x := Q_O2[ok])))*(100 < x)
27   Q_O2[ok] += qty
```

Figure 3.3 – The insert trigger program for the simplified TPC-H Q18.

shows the generated trigger program. The AGCA expression Q for the query is:

$$\text{Sum}_{[CK]}\big(C(CK) * O(CK,OK) * LI(OK,QTY) * QTY * (x := Q_n) * (100 < x)\big)$$

where $Q_n = \text{Sum}_{[]}\big(LI(OK_1,QTY_1) * (OK = OK_1) * QTY_1\big)$. First, we simplify the subquery Q_n. Unification eliminates the equality predicate to yield an expression with no input variables: $Q'_n = \text{Sum}_{[OK]}\big(LI(OK,QTY_1) * QTY_1\big)$. Next, we show the derivation of deltas for insertions into Orders O, Lineitem LI, and Customer C. The deletion deltas for all relations are duals of the insertion deltas, so we omit them entirely.

Insertions into Orders

The first-order delta $\Delta_{+O(ck,ok)}Q$ for the insertion of a single tuple $\langle CK : ck, OK : ok\rangle$ is:

$$\text{Sum}_{[CK]}\big(C(CK) * LI(OK,QTY) * (OK := ok) * (CK := ck) * QTY * (x := Q'_n) * (100 < x)\big)$$

The delta expression gets simplified after propagating the assignments: every occurrence of OK and CK is replaced with ok and ck, respectively; these assignments are also safe to remove. The delta expression is $\text{Sum}_{[ck]}(C(ck) * LI(ok, QTY) * QTY * (x := Q'_n) * (100 < x))$ with OK replaced by ok inside Q'_n.

Query decomposition splits the delta into three parts: $C(ck)$ has no common columns with the rest of the expression and is materialized as a separate map. The remaining expression can also be divided into two subexpressions that share only the trigger variable ok. Then, since the selection predicate is being applied to a singleton, we can safely materialize only the aggregate in the assignment. Applying these optimizations yields the following materialization decision:

$$\mathcal{M}\big(\text{Sum}_{[ck]}(C(ck))\big) * \mathcal{M}\big(\text{Sum}_{[ok]}(LI(ok, QTY) * QTY)\big) * \text{Sum}_{[]}\big((x := \mathcal{M}(Q'_n)) * (100 < x)\big)$$

The trigger statement uses the following views (Figure 3.3, line 15, note that Q_{O2} is used twice): $Q_{O1}[CK] := \text{Sum}_{[CK]}\big(C(CK)\big)$ and $Q_{O2}[OK] := \text{Sum}_{[OK]}\big(LI(OK, QTY) * QTY\big)$. The delta for $Q_{O1}[CK]$ and insertions into C is $\Delta_{+C(ck)} Q_{O1} := \{\langle CK : ck\rangle \mapsto \langle 1, 1\rangle\}$, which corresponds to trigger statement 12. $Q_{O2}[OK]$ is maintained similarly with trigger statement 27.

Insertions into Lineitem

With the nested subquery correlated on an equality, Higher-Order IVM chooses to compute the first-order delta of Q for the insertion of a single tuple $\langle OK : ok, QTY : qty\rangle$. The revised rule for nested subqueries yields:

$$\Delta_{+LI(ok,qty)}(x := Q'_n) \;:=\; (OK := ok) * \big((x := Q'_n + qty) - (x := Q'_n)\big)$$

Following the delta rule for products

$$\Delta_{+LI(ok,qty)} Q \;:=\; \text{Sum}_{[CK]}\big(C(CK) * O(CK, OK) * \Delta Q_{LI} * QTY * (100 < x)\big)$$

where

$$\begin{aligned}\Delta Q_{LI} \;:=\; & (OK := ok) * \big((QTY := qty) * (x := Q'_n) + \\ & LI(OK, QTY) * \big((x := Q'_n + qty) - (x := Q'_n)\big) + (QTY := qty) * \big((x := Q'_n + qty) - (x := Q'_n)\big)\big)\end{aligned}$$

.
Polynomial expansion, partial evaluation, and unification result in:

$$\begin{aligned}\Delta_{+LI(ok,qty)} Q := \; & \text{Sum}_{[CK]}\big(C(CK) * O(CK, ok) * (LI(ok, QTY) * QTY * (x := Q'_n + qty) - \\ & LI(ok, QTY) * QTY * (x := Q'_n) + qty * (x := Q'_n + qty)) * (100 < x)\big)\end{aligned}$$

Decomposition and polynomial expansion let us extract the terms $\text{Sum}_{[CK,ok]}(C * O)$ and $\text{Sum}_{[ok]}(LI(ok, QTY) * QTY)$ as separate maps. The rewrite rules for nested aggregates and input variables materialize Q'_n. The final materialization is:

$$\mathcal{M}\big(\mathrm{Sum}_{[CK,ok]}(C(CK) * O(CK,ok))\big) * \big(\mathcal{M}(Q_2) * (x := \mathcal{M}(Q_n') + qty) - \mathcal{M}(Q_2) * (x := \mathcal{M}(Q_n')) + qty * (x := \mathcal{M}(Q_n') + qty)\big) * (100 < x)$$

where $Q_2 = \mathrm{Sum}_{[ok]}\big(LI(ok,QTY) * QTY\big)$.

Apart from the outermost materialization (of $C \bowtie O$), the remaining five materialized maps are identical to Q_{O2}, which is already being maintained. Thus, only one additional view, $Q_{LI}[CK,OK] := \mathrm{Sum}_{[CK,OK]}(C(CK) * O(CK,OK))$, has to be maintained. Rewriting the materialization decision produces trigger statements 21–23. Q_{LI} is maintained analogously as in Example 2.2.5, resulting in trigger statements 11, 12, 16, and 17.

Note that statements 21–23, which update Q, include a loop. However, even though our implementation is not explicitly aware of TPC-H's foreign key dependencies, in this example, only one customer (CK) will be updated.

Insertions into Customer

The first-order delta $\Delta_{+C(ck)}Q$ for the insertion of a single tuple $\langle CK : ck \rangle$ is

$$\mathrm{Sum}_{[CK]}\big((CK := ck) * LI(OK,QTY) * OK(CK,OK) * QTY * (x := Q_n') * (100 < x)\big)$$

After applying unification, we materialize the entire delta expression as Q_C. We leave the derivation of the remaining trigger statements as an exercise for the reader.

3.3.2 The Pricespread Query (PSP)

Next, we look at the query PSP from our test workload, which has two nested aggregates:

```
SELECT SUM(A.P - B.P) FROM A, B
WHERE B.V > (SELECT SUM(B'.P * 0.0001) FROM B AS B') AND
      A.V > (SELECT SUM(A'.P * 0.0001) FROM A AS A')
```

Again, for simplicity, we use the condensed schema $B(P,V)$ and $A(P,V)$. The AGCA expression Q for the query is:

$$\begin{aligned}\mathrm{Sum}_{[]}\big(&B(BP,BV) * A(AP,AV) * (AP - BP) * \\ &(v1 := \mathrm{Sum}_{[]}(B(BP',BV') * BV' * 0.0001)) * (BV > v1) * \\ &(v2 := \mathrm{Sum}_{[]}(A(AP',AV') * AV' * 0.0001)) * (AV > v2)\big)\end{aligned}$$

Figure 3.4 shows the trigger program. Since the aggregates have no correlated variables, they can be decorrelated. Subsequently, there is no benefit to using deltas to update the final query result and our compilation heuristics decide on full recomputation for updates to both A and B. Hence, rather than describing the full compilation process for this example, we focus on the process of materializing the full query.

Listing 3.4 Single-tuple triggers for the PriceSpread query

```
1  Top-level and 6 auxiliary materialized views (3 are omitted):
2    Q[] := (B(BP,BV)*A(AP,AV)*(AP-BP)*
3          (v1 := Sum[](B(BP',BV')*BV'*0.0001))*(BV>v1)*
4          (v2 := Sum[](A(AP',AV')*AV'*0.0001))*(AV>v2)
5    Q_B1[BV] := Sum_[BV](B(BP,BV)*BP)
6    Q_B2[]   := Sum_[](B(BP,BV)*BV)
7    Q_B3[BV] := Sum_[BV](B(BP,BV))
8
9  on insert into B values (bv,bp):
10   Q_B1[bv] += bp
11   Q_B2[]   += bv
12   Q_B3[bv] += 1
13   Q[] := (Sum_[]((v1 := Q_B2[]*0.0001) * Q_B3[BV'] * (BV'>v1)) *
14           Sum_[]((v2 := Q_A2[]*0.0001) * Q_A1[AV'] * (AV'>v2))) -
15          (Sum_[]((v1 := Q_B2[]*0.0001) * Q_B1[BV'] * (BV'>v1)) *
16           Sum_[]((v2 := Q_A2[]*0.0001) * Q_A3[AV'] * (AV'>v2)))
```

Figure 3.4 – The trigger program for PSP insertions into B. The deletion trigger for B and the triggers for A are symmetric.

The join graph of this expression is intriguing. It consists of two mostly disconnected, symmetric components, one for $B(BP,BV)$ and one for $A(AP,AV)$. In fact, the only edge between these two is the term $(AP-BP)$. Our materialization strategy exploits both this, and the fact that integer addition and bag union are identical in AGCA.

Starting with the default materialization strategy $\mathcal{M}(Q)$, we perform polynomial expansion (Rule 2). Because AGCA does not separate integer addition from bag union, this distributes the rest of the expression over the term $(AP-BP)$.

$$\begin{aligned}
&\mathcal{M}\big(\mathrm{Sum}_{[]}\big(B(BP,BV)*A(AP,AV)*AP*(v1:=\mathrm{Sum}_{[]}(B(BP',BV')*BV'*0.0001))*\\
&\qquad (v2:=\mathrm{Sum}_{[]}(A(AP',AV')*AV'*0.0001))*(BV>v1)*(AV>v2)\big)\big)-\\
&\mathcal{M}\big(\mathrm{Sum}_{[]}\big(B(BP,BV)*A(AP,AV)*BP*(v1:=\mathrm{Sum}_{[]}(B(AP',BV')*BV'*0.0001))*\\
&\qquad (v2:=\mathrm{Sum}_{[]}(A(AP',AV')*AV'*0.0001))*(BV>v1)*(AV>v2)\big)\big)
\end{aligned}$$

We can now decorrelate the nested aggregates (Rule 4). This expression contains two identical aggregates, each computing the total volume of B or A. We we call these Q_{B2} and Q_{A2}. As only one relation appears in each aggregate, maintenance requires only a single statement each, shown in the trigger program for B as statement 11.

$$\begin{aligned}
&\mathrm{Sum}_{[]}\big(\mathcal{M}\big(\mathrm{Sum}_{[\mathrm{BV,AV}]}(B(BP,BV)*A(AP,AV)*AP)\big)*\\
&\qquad (v1:=Q_{B2}[]*0.0001)*(BV>v1)*(v2:=Q_{A2}[]*0.0001)*(AV>v2)\big)-\\
&\mathrm{Sum}_{[]}\big(\mathcal{M}\big(\mathrm{Sum}_{[\mathrm{BV,AV}]}(B(BP,BV)*A(AP,AV)*BP)\big)*\\
&\qquad (v1:=Q_{B2}[]*0.0001)*(BV>v1)*(v2:=Q_{A2}[]*0.0001)*(AV>v2)\big)
\end{aligned}$$

After polynomial expansion the expression computes two joins instead of one. The hypergraphs of the simpler joins, however, contain disconnected components. We can apply decomposition (Rule 1) to each.

$$\begin{aligned}
&\mathrm{Sum}_{[]}\big(\,\mathcal{M}\big(\mathrm{Sum}_{[\mathrm{BV}]}(B(BP,BV))\big) * \mathcal{M}\big(\mathrm{Sum}_{[\mathrm{AV}]}(A(AP,AV) * AP)\big) * \\
&\qquad (v1 := Q_{B2}[] * 0.0001) * (BV > v1) * (v2 := Q_{A2}[] * 0.0001) * (AV > v2)\,\big) - \\
&\mathrm{Sum}_{[]}\big(\,\mathcal{M}\big(\mathrm{Sum}_{[\mathrm{BV}]}(B(BP,BV) * BP)\big) * \mathcal{M}\big(\mathrm{Sum}_{[\mathrm{AV}]}(A(AP,AV))\big) * \\
&\qquad (v1 := Q_{B2}[] * 0.0001) * (BV > v1) * (v2 := Q_{A2}[] * 0.0001) * (AV > v2)\,\big)
\end{aligned}$$

Now, no further rules are applicable. We materialize four additional maps: for each volume we maintain both the count and sum of prices of both relations. In the trigger program, maps Q_{B3} and Q_{A3} maintain the counts using statement 12 and its dual in A; maps Q_{B1} and Q_{A1} maintain the price sums using statements 10 and its dual in A.

Because the total volume of each relation changes with every insertion, we must recompute the price and count totals for the relation that changes. Specialized data structures such as range trees could further reduce the cost of doing so by allowing us to efficiently maintain expressions of the form $\mathcal{M}\big(\mathrm{Sum}_{[\mathrm{BV}]}\big(B(BP,BV) * (BV > v1)\big)\big)$. Nevertheless, by exploiting the connection between addition and bag union, we evaluate this expression using only scans (in contrast to first computing a Cartesian product as a traditional database system would).

3.4 Efficient Delta Evaluation for Batch Updates

So far, we have focused mostly on single-tuple updates, but the viewlet transform and many of the presented optimizations also support batches of updates. In batch delta processing, replacing large base relations by much smaller deltas reduces evaluation cost, favoring incremental maintenance over recomputation.

In this section, we focus on the efficient evaluation of delta queries for batches of updates to base relations. In particular, we consider queries with nested aggregates and existential quantification for which the delta rule for variable assignment prescribes recomputing both the old and new results to evaluate the delta – that clearly costs more than recomputing the whole query expression once. In general, these classes of queries might have no benefit from incremental maintenance. But, in many cases, we can specialize this delta rule to achieve efficient maintenance, as we demonstrated for single-tuple updates in Section 3.2.3. Here, we generalize this approach to both single-tuple and batch updates.

3.4.1 Model of Computation

We describe our model of computation to understand the advantages of alternative evaluation strategies. We represent an expression as a tree of operators, which are always evaluated from left to right, in a bottom-up fashion. Information about bound variables flows from left to

right through the product operation. For instance, in expression $R(A) * S(A)$, the term $R(A)$ binds the A variable which is then used to lookup the multiplicity value inside $S(A)$. The evaluation cost for such an expression is $\mathcal{O}(|R|)$, where $|R|$ is the number of tuples with a non-zero multiplicity in R.

Our model considers in-memory hash join as a reference join implementation. In this model, the ordering of terms has an impact on query evaluation performance. For example, when $S(A)$ is smaller than $R(A)$, commuting the two terms like $S(A) * R(A)$ results in fewer memory lookups in R. Note that commuting terms is not always possible – for instance, in expression $R(A) * A$, the two terms do not commute, unless A is already bound.

3.4.2 Domain extraction

We present a technique, called *domain extraction*, for efficient delta computation of queries with nested aggregates and existential quantification. We explain the idea on the problem of duplicate elimination in bag algebra. We formalize the technique afterwards.

For clarity of the presentation, we introduce Exist(Q) as syntactic sugar for $\text{Sum}_{[\text{sch(Q)}]}((X := Q) * (X \neq 0))$, where $sch(Q)$ denotes the schema of Q. Exist(Q) changes every non-zero multiplicity inside Q to 1. The delta rule for Exist is $\Delta_R(\text{Exist}(Q)) = \text{Exist}(Q + \Delta_R Q) - \text{Exist}(Q)$.

First, we introduce the notion of domain expressions. A domain expression binds a set of variables with the sole purpose of speeding up the downstream query evaluation. All domain expression tuples have the multiplicity of one. For instance, we can write $R(A,B)$ as $\text{Exist}(R(A,B)) * R(A,B)$ without changing the original query semantics. Note that now $\text{Exist}(R(A,B))$ defines the iteration domain during query evaluation rather than $R(A,B)$.

Example 3.4.1 Consider an SQL query over $R(A,B)$

```
SELECT DISTINCT A FROM R WHERE B > 3
```

or equivalently $Q := \text{Exist}(\text{Sum}_{[\text{A}]}(R(A,B) * (B > 3)))$. Let Q_n denotes the nested sum, then $\Delta Q_n := \text{Sum}_{[\text{A}]}(\Delta R(A,B) * (B > 3))$ and $\Delta Q := \text{Exist}(Q_n + \Delta Q_n) - \text{Exist}(Q_n)$. Note that ΔQ recomputes Q twice, once to insert new tuples and then to delete the old ones, which clearly defeats the purpose of incremental computation. Also, $(Q_n + \Delta Q_n)$ might leave unchanged many tuples in Q_n, so deleting those tuples and inserting them again is wasted work.

Our goal is to transform ΔQ into an expression that changes only relevant tuples in the delta result. We want to iterate over only those tuples in Q_n whose A values appear in $\Delta R(A,B)$; other tuples are irrelevant for computing ΔQ. To achieve that goal, we capture the domain of A values in ΔR using $\text{Exist}(\Delta R(A,B))$[2]. To express only distinct values of A in ΔR, we write:

$$Q_{dom} := \text{Exist}(\text{Sum}_{[\text{A}]}(\text{Exist}(\Delta R(A,B)))).$$

[2] ΔR can contain both insertions and deletions.

Algorithm 3.5 Domain Extraction

```
def extractDom(e: Expr): Expr = e match {
  case Plus(A, B) =>
    interDoms(extractDom(A), extractDom(B))
  case Prod(A, B) =>
    unionDoms(extractDom(A), extractDom(B))
  case Sum(gb, A) =>
    val domA  = extractDom(A)
    val domGb = inter(sch(domA), gb)
    if (domGb == gb) domA
    else if (domGb == Nil) 1
    else Exist(Sum(domGb, domA))
  case Assign(v, A) if A.hasRelations =>
    extractDom(A)
  case Rel(_)   =>
    if (e.hasLowCardinality) Exist(e) else 1
  case _        => e
}
```

Figure 3.5 – The domain extraction algorithm. `interDoms` extracts common domains, `unionDoms` merges domains, `inter` is set intersection, and `sch(A)` is the schema of `A`.

The outer Exist keeps the multiplicity of 1. We prepend Q_{dom} to ΔQ to restrict the iteration domain.

$$\Delta Q' := Q_{dom} * (\text{Exist}(Q_n + \Delta Q_n) - \text{Exist}(Q_n))$$

Note that we can make Q_{dom} more strict by including $(B > 3)$. □

Figure 3.5 shows the algorithm for domain extraction. The algorithm recursively pushes extracted domains up through the expression tree to bound variables in even larger parts of the expression. At leaf nodes, it identifies relations with low cardinalities that can restrict the iteration domain. We can either use cardinality estimates for each relation or rely on heuristics. We also include terms that can further restrict the domain size, like comparisons, values, and variable assignments over values.

For Sum aggregates, we recursively compute the domain of the subexpression and then, if necessary, reduce its schema to match that of the Sum aggregate. For instance, in Example 3.4.1, we project $\text{Exist}(\Delta R(A,B)) * (B > 3)$ on column A. If the domain schema is reduced, we enclose the expression with Exist to preserve the domain semantics; if the schema is empty, the extracted domain bounds no column and has little effect. When dealing with union, we intersect two subexpressions to find the maximum common domain to be propagated further up in the tree; for the product operation, we union subexpressions into one common domain.

The domain extraction procedure allows us to revise the delta rule for variable assignments as:

$$\Delta(var := Q) := Q_{dom} * ((var := Q + \Delta Q) - (var := Q))$$

where $Q_{dom} :=$ `extractDom`(ΔQ).

3.4.3 Single-tuple vs. Batch Updates

Incremental programs for single-tuple updates are simpler and easier to optimize than batched incremental programs. The parameters of single-tuple triggers match the tuple's schema to avoid boxing and unboxing of the primitive data types in the input. We can inline these parameters into delta expressions and eliminate one-element loops. In local environments, we can process one batch of updates via repeated calls to single-tuple triggers. Aside from the invocation overhead, we identify three reasons why such an approach can be suboptimal.

Preprocessing batches Preprocessing a batch of updates can merge or eliminate changes to the same input tuples. Static analysis of the query can identify subexpressions that involve solely batch updates (e.g., domain expressions). Then, we can precompute a batch aggregate by keeping only relevant batch columns of the tuples that match query's static conditions. Batch pre-aggregation can produce much fewer tuples, which can significantly decrease the cost of trigger evaluation; smaller batches also reduce communication costs in distributed environments. Batch pre-aggregation can have a limited effect when there is no filtering condition and there is a functional dependency between the aggregated columns and the primary key of the delta relation; then, we can eliminate only updates targeting the same key.

Skipping intermediate views states When a maintenance trigger evaluates the whole query from scratch, batching can help us to avoid recomputation of intermediate query results. For instance, considering the query of Example 3.5.1 and updates to S, processing one batch of updates refreshes the inner query result once and triggers one recomputation of the outer query. In contrast, the tuple-at-a-time approach evaluates the outer query on every update.

Cache locality Processing one update batch, in a tight loop, can improve cache locality and branch prediction for reasonably-sized batches; too large batches can have negative impacts on locality.

In the experimental evaluation in Chapter 4, we evaluate these trade-offs between single-tuple and batch maintenance for different update sizes.

3.5 Re-evaluation vs. Incremental Computation

In certain cases, re-evaluation from scratch is preferable to incremental computation. For some queries involving comparisons on nested aggregate values, computing the delta query can be more expensive than recomputing the original query. This is typically the case if there is no equality correlation between the nested subquery and the outer query. The first decision the optimizer makes is whether to use the delta query to update the materialized view (i.e., $Q[\vec{X}]$ += $\Delta_u Q[\vec{Y}]$) or to recompute the materialized view from scratch (i.e., $Q[\vec{X}]$:= $Q[\vec{Y}]$).

Example 3.5.1 Consider a query over $R(A,B)$ and $S(B,C)$.

```
SELECT COUNT(*) FROM R
WHERE R.A < (SELECT COUNT(*) FROM S) AND R.B = 10
```

The nested query computes an aggregate value and has no correlation with the outer query. The domain extraction procedure cannot restrict the domain of the delta query for updates to S, so re-evaluation is a better option. We can still accelerate the computation by materializing the query piecewise, for instance, by precomputing and maintaining the expression $\text{Sum}_{[A]}(R(A,B) * (B = 10))$. Note that we can incrementally maintain the top-level query for updates to R by materializing the nested query result as a single variable. □

Nested queries are often correlated with the outside query. When the correlation involves equality predicates, extracting the domain of the inner query might restrict some of the correlated variables. This range restriction can reduce the maintenance cost. In general, the decision on whether to incrementally maintain or recompute the query result requires a case-by-case cost analysis. Our default heuristic rule decides to incrementally maintain a query whenever the extracted nested domain binds at least one equality-correlated variable.

If one decides to recompute the expression Q, the materialization strategy must avoid creating a self-referential loop. The default materialization decision $\mathcal{M}(Q)$ is meaningless as Q defines the view being maintained. In other words, this strategy has no materialized subviews than can be used to recompute Q. We avoid this problem by first applying the nested-query rewrite heuristic (Rule 4) as aggressively as possible. Because recomputation is only appropriate for queries with nested subqueries, the resulting expression is guaranteed to be simpler.

3.6 Cost-Based Optimization

Manual inspection of most materialization plans generated for our test workload suggests that a purely heuristic materialization optimizer is typically sufficient to achieve near-optimal results. However, there are some cases where a more thorough cost-based analysis is required. This is especially the case where the optimal strategy depends on data-specific parameters such as the distribution of data values.

Concretely, our cost-based optimization strategy explores a search space of materialization strategies for each statement $Q[\vec{X}]$ += $\Delta_u Q[\vec{Y}]$, defined along 3 dimensions: (1) Depth of Incremental Computation, (2) The Materialization Decision, and (3) Specialized Data Structures.

Using the simple cost model described below and starting with the materialization decision generated by the heuristic strategy, a simple greedy gradient descent algorithm suffices to efficiently explore the (exponential) search space of possibilities and provide a near-optimal solution. More potent search space exploration techniques (e.g., MCMC methods like Metropolis-Hastings or Simulated Annealing) are also possible but beyond the scope of this work. Also note that queries in our scenarios are expected to be long-lived, so, it is often reasonable to perform an exhaustive exploration of the full search space for such queries.

3.6.1 Depth of Incremental Computation

For flat (non-nested) queries, Higher-Order IVM has lower parallel complexity than non-incremental evaluation [94, 93], which often translates in faster execution on sequential machines. But, in certain cases, the materialized views generated via the full recursive compilation can incur substantial maintenance costs, leading to Higher-Order IVM to lose its performance advantage over classical IVM. For instance, in a query enumerating all triangles in a graph, the full recursive approach stores and maintains all 2-length paths (in addition to the base edge relation). The number of such paths can be much larger than the number of triangles in the graph ($\mathcal{O}(N^2)$ vs. $\mathcal{O}(N^{1.5})$ in the worst case), suggesting that classical IVM is a better option here. Motivated by recent developments in the theory of worst-case optimal joins [144, 118], our heuristic rule for cyclic queries is to materialize only cyclic sub-queries and then use these sub-views to maintain the top-level query.

For queries with nested aggregates, re-evaluation might be preferable to incremental computation, as discussed in Section 3.5. When domain extraction in a delta query is possible (i.e., there is an equality-based correlation between the inner and outer query), we have to choose between re-evaluation (depth-0 compilation) and incremental computation (depth-1 classical IVM or Higher-Order IVM). The decision on the evaluation strategy is guided by the estimated size of the extracted domain (small domain: incremental, large domain: re-evaluation).

3.6.2 The Materialization Decision

We define the the search space for materialization decisions locally, using two rewrite rules. The polynomial factorization/expansion rule (Figure 3.1.2) can be applied in either direction. In addition to factorizing or expanding polynomial terms in queries, this rule also allows the materialization operator to be pushed into or pulled out of addition and summation terms.

The second rewrite rule is a generalization of the query decomposition rule (Figure 3.1.1) as follows. First, we make the rule bi-directional, so it can push or pull the materialization operator into or out of product terms. Second, we relax the restriction on the intersection of variable sets $\vec{A}$ and $\vec{B}$ so that the rule can be applied to any product of terms. Note that this generalized decomposition rule also subsumes the input variable rewriting rule (Figure 3.1.3).

3.6.3 Specialized Data Structures

Thus far, we have considered only straightforward view materialization where views are stored in map-like data structures. But for some queries, advanced data storage primitives can provide opportunities to materialize more complex expressions — particularly those involving input variables. As many of these opportunities are data-dependent, we can rely on user input to direct selection of an appropriate data structure, or we can use a cost-based tuning advisor to automate the selection process with minimal user involvement.

One example of a specialized data structure is *view cache*, which materializes AGCA expressions with input variables. A view cache stores multiple full copies of the materialized view, each for a different valuation of the input variable(s) that appear in the cache's defining expression. When a lookup is performed on the cache, these input variables must be bound to specific values. If the cache contains a materialized view for that particular valuation, the materialized view is returned as a normal map. Otherwise, the cache's defining query is evaluated as normal, and the result is stored in the cache. Unlike a traditional cache, the contents of a view cache is never invalidated. Instead, whenever the underlying data is changed, each materialized view stored in the view cache is updated as normal.

View caches are only beneficial when the domain of an input variable is small. Otherwise, they can incur significant maintenance and memory overheads, and the heuristic optimizer should refrain from creating them.

3.6.4 Cost Model

We now discuss the cost model for making materialization decisions. The model relies on standard cardinality estimation techniques [61, 146] for a first-pass approximation of execution costs and estimates of the relative update event and query result request frequencies.

Like the viewlet transform itself, the cost model is recursive: The cost of a materialization decision is dependent on the cost of maintaining each materialized view under it. Consequently, for a materialization decision $\langle Q', \vec{M}_Q \rangle$, we make a distinction between the *execution cost* of Q', and the *maintenance cost* of each materialized view M_Q.

Update Rates

The trigger program generated by a viewlet transform is event-based. When evaluating the cost of a viewlet transform, we must consider the relative rates at which these events occur. Concretely, we consider: (1) The rate $\rho_{\pm R}$ at which each relation update event $\pm R$ occurs, and (2) The rate ρ_Q at which the materialized query results are requested. Note that we are only concerned with the relative rates at which these events will occur. Any unit of rate may be selected arbitrarily for these values, as long as it is used consistently.

For some application domains, these two rates may be interrelated. For example, if the original query encodes a constraint being enforced over the underlying data, then the constraint will need to be validated on every update, and $\rho_Q = \sum_{\pm R} \rho_{\pm R}$. Conversely, if the view is being maintained to reduce lookup latencies, then ρ_Q is defined by the application. In the absence of user-provided statistics, we assume a uniform distribution of update rates (i.e., $\rho_{\pm R} = 1$), and a query request on every update (i.e., $\rho_Q = \sum_{\pm R} \rho_{\pm R}$).

Evaluation Cost

Through benchmarking of our experimental workload, we have found that the dominant cost of query evaluation comes from materializing results, both intermediate and final. Consequently, we express the evaluation cost in units of number of tuples materialized.

Although our implementation materializes intermediate results only for the $\text{Sum}_{[]}$ operator, the cost estimation strategy we will describe can be easily generalized to more complex query evaluation strategies. The cost of materializing a $\text{Sum}_{[]}$ is proportional to the cardinality of the query being aggregated. Similarly, the cost of materializing the result of an update query is proportional to the cardinality of its output.

We use standard estimation techniques [61, 146] to estimate the cardinality $|Q|$ of query Q. The full execution cost of the query Q is its cardinality, plus the cost of materializing each intermediate result.

$$\texttt{cost}_{\texttt{exec}}(Q) = |Q| + \sum_{\text{Sum}_{[]}(Q_i) \in Q} |Q_i|$$

Example 3.6.1 Consider the query computing a single aggregate over a join between R and T.

$$Q := S(A) * \text{Sum}_{[A]}(R(A,B) * T(B,C))$$

The cost of materializing the intermediate result for this aggregate is $|R(A,B) * T(B,C)|$. The cost of materializing the final $|Q|$, and so: $\texttt{cost}_{\texttt{exec}}(Q) = |Q| + |R(A,B) * T(B,C)|$. □

We target application domains where the size of the base relations is relatively stable over the course of execution (e.g., as in our financial workload) or application domains where delta query evaluation cost is independent of the size of the base relations (as in many TPC-H queries). In either case, we can obtain a steady-state relation size (which is appropriate for the first case, and ignored by the latter) for cardinality estimation.

Maintenance Cost

We are now ready to discuss the cost of maintaining materialized views. Unlike query execution, maintenance is an ongoing process. Consequently, maintenance costs are expressed in terms of costs and the rates at which costs are incurred.

Maintaining a view defined by query Q requires repeatedly evaluating the view's delta queries when the corresponding update events occur, and the delta query $\Delta_{\pm R}Q$ will be evaluated at a rate of $\rho_{\pm R}$. Naïvely, the total maintenance cost of this view could be expressed in terms of the rate-weighted cost of each delta query.

$$\sum_{\pm R} \rho_{\pm R} \cdot \texttt{cost}_{\texttt{exec}}(\Delta_{\pm R}Q)$$

However, this expression does not account for the cost of maintaining views created to support evaluation of $\Delta_{\pm R}Q$. Consequently, we take a holistic approach to cost estimation.

For a query Q, we consider each trigger $\pm R$ generated by the viewlet transform of Q. The total execution cost for the trigger is the sum of execution costs of each statement $M_{\pm R,i} += Q_{\pm R,i}$. The total maintenance cost of Q is the rate-weighted cost of each trigger.

$$\sum_{\pm R}\left(\rho_{\pm R}\cdot\sum_{i}\texttt{cost}_{\texttt{exec}}(Q_{\pm R,i})\right)$$

Finally, we consider the rate at which query Q is requested. If the value of Q is requested at rate ρ_Q, then the full cost of maintaining Q using materialization decision $\langle Q', \vec{M}\rangle$ is

$$(\rho_Q\cdot\texttt{cost}_{\texttt{exec}}(Q))+\sum_{\pm R}\left(\rho_{\pm R}\cdot\sum_{i}\texttt{cost}_{\texttt{exec}}(Q_{\pm R,i})\right)$$

3.7 Summary

The naïve viewlet transform described in Chapter 2 is too aggressive in some cases, like with inequality joins and certain nesting patterns, for which parts of queries are better re-evaluated than incrementally maintained. We develop heuristic rules for trading off between materialization and lazy evaluation for the best performance. We present the domain extraction procedure for maintaining queries with equality-correlated nested aggregates, which no other commercial database currently supports. We also discuss the trade-offs between re-evaluation and incremental computation, and we identify the cases when batching can significantly reduce maintenance costs compared to single-tuple execution. Finally, we present a cost-based model for deciding on materialization decisions.

4 The DBToaster System

The DBToaster system[1] implements Higher-Order Incremental View Maintenance.

DBToaster is a compiler for database queries. It transforms long-lived (continuous) queries into high-performance stream processing engines that keep query results (views) fresh at very high update rates. The compiler relies on Higher-Order IVM and code generation to produce efficient update triggers for maintaining materialized views in main memory.

DBToaster targets applications that require real-time, low-latency processing over changing datasets. It combines the advantages of data stream processing engines and relational database systems. Like stream processors, it supports continuous queries and keeps their results always fresh but without restrictive window semantics; like relational databases, it provides rich support for analytical queries using the SQL language but offers much lower view refresh latencies. DBToaster enables developers to build real-time engines directly from their declarative specifications, which can significantly improve their productivity and eliminate the need for building ad hoc solutions for performance reasons.

4.1 System Overview and Application Usage

The DBToaster compiler consists of two parts. The frontend converts SQL queries into AGCA expressions and applies Higher-Order IVM to produce an intermediate trigger representation describing how to maintain the query results. The backend optimizes these triggers and transforms them into source code. Currently, DBToaster can generate imperative (C++) and functional (Scala) code for local execution and data-parallel code (Spark) for distributed execution. The compiler also comes with a runtime library that provides data loading and instrumentation of generated query engines.

The DBToaster compiler produces query processors that are aggressively specialized to a specific query workload, rather than ad-hoc queries. Created engines are stand-alone and

[1] DBToaster is available for download at www.dbtoaster.org.

require no other system (e.g., a database system) to run. End-users interact with a DBTOASTER-generated query processor in one of three ways:

1. *Standalone binaries*, where users may run the binary on a file or specify a listening socket through which data can be sent to the engine. The engine can output a stream of view query results to a file or a network connection.

2. *Shared libraries*, where application developers may link against our library and directly access to snapshots of the in-memory data structures representing our views while they are concurrently maintained. We are exploring asynchronous notification methods to support push-based application logic, including callback functions and futures registered with our views.

3. *Source code*, where application developers may adapt and extend our query processing engine as desired, for example to use custom data structures to implement views.

DBTOASTER produces extensible query engines capable of custom stream pre-processing and workload generation, as well as on-demand querying of views. Our object-oriented design enables users to inherit our engine in their applications, where they may override a pre-processing method invoked on each arriving event. This allows users to perform basic data extraction, transformation, cleaning, and logging functionality prior to delta processing.

DBTOASTER can generate query engines for running in both local and distributed settings. Generated local code implements a single-core, single-threaded query executor. Our implementation strategy has focused on novel view maintenance techniques rather than the full range of state-of-the-art query execution mechanisms. Thus, our results represent a lower limit on performance and scalability, both of which one could substantially improve with multithreaded and vectorized execution that utilizes more flexible view data structures as inspired by database cracking [83]. Chapter 5 presents DBTOASTER's distributed main-memory engine that exploits aggregate CPU and memory resources available in large-scale clusters to achieve low-latency incremental view maintenance.

4.2 Implementing View Maintenance

In this section, we describe how DBTOASTER compiles AGCA queries into specialized code and data structures. While we could directly interpret trigger programs, for efficient execution, we translate them into lower-level code. Our approach brings programming language optimizations to query processing, enabling holistic query optimization and generation of high-performance engines. For comparison, traditional database systems offer general-purpose query processing engines capable of handling arbitrary query workloads. Nowadays, however, databases process mostly parametrized queries, for which the use of generic storage structures and template-based operators is suboptimal [116, 92].

4.2.1 From Queries to Native Code

DBToaster compiles intermediate trigger programs into source code optimized for the execution in local mode (C++ and Scala) or distributed mode (Scala for Spark). The compiler uses in-memory hash join as a reference join implementation and assumes a computation model in which the information about bound variables always flows from left to right during query evaluation. This notion of information flow eases compilation of the AGCA language construct. In generated code, we replace all high-level operators, like natural joins, unions, etc., with concrete operations over the underlying data structures.

When compiling a relational term, we distinguish several cases: (1) if all its variables are free, we transform it into a `foreach` loop that iterates over the whole collection; (2) if all its variables are bound, we replace it with a `get` (lookup) operation; (3) otherwise, we form a `slice` operation that iterates over only the elements matching the current values of the bound variables. Note that cases (2) and (3) may benefit from specialized index structures.

We use continuation passing style [31] to facilitate code generation and avoid intermediate materializations, such as redundant computations of bag unions and aggregates. The compilation process relies on an extensible compiler library [136, 130] to build an abstract representation of the program and perform both traditional compiler optimizations, like common subexpression elimination, dead code elimination, aggressive inlining, loop unrolling and fusion, and also domain-specific optimizations, like data-structure specialization and automatic indexing. DBToaster's code generators emit source code directly from such an optimized representation.

DBToaster optimizes incremental programs for single-tuple processing. It specializes the parameters of single-tuple triggers to the concrete primitive types of the updated relation. Then, the native compiler can treat such parameters as constants and move them out of loops and closures when possible. Our compiler eliminates loops around one-element batches and uses primitive type variables rather than maps to store intermediate materializations.

4.2.2 Data Structure Specialization

The design of the data structure for storing materialized views depends on whether the view contents can change over time and the types of operations that need to be supported. Materialized views defined over static base relations are immutable collections of records stored in fixed-size arrays with fast lookup and scan operations. In typical streaming scenarios, however, updates to mutable base relations can be fast-moving and unpredictable, indicating that any array-based solution for storing tuples of dynamic materialized views might be expensive either in terms of the memory usage or maintenance overhead.

We materialize the contents of dynamic materialized views inside record pools, shown in Figure 4.1. One record pool stores records of the same format inside main memory and dynamically adapts its size to match the current working set. The pool keeps track of available

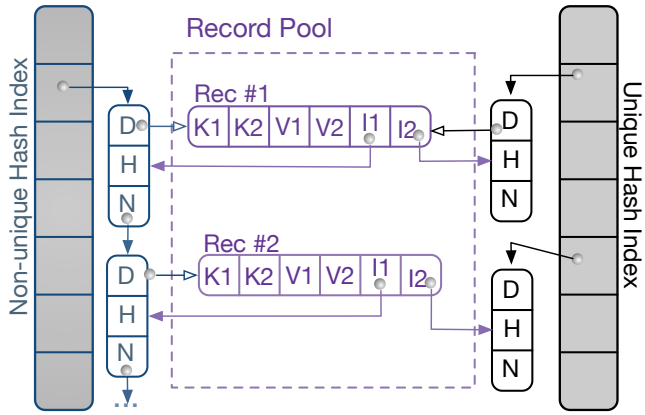


Figure 4.1 – Multi-indexed data structure used for materialization. Each bucket (shaded cells) has a linked list of collisions. Legend: (D)ata, (H)ash, (N)ext, (K)ey, (V)alue, and (I)ndex.

free slots to facilitate future memory allocations and ease memory management.We specialize the format of pool records at compile time. Each record contains key fields corresponding to the schema of the materialized expression and value fields storing tuple multiplicities. The key fields uniquely identify each record.

Automatic Index Support

We can associate multiple index structures with one record pool, as shown in Figure 4.1. Each index structure provides a fast path to the records that match a given condition. We use unique hash indexes to provide fast lookups and non-unique hash indexes for slice operations[2]. Both indexes maintain an overflow linked list for each bucket. To shorten scanning in one bucket, non-unique hash indexes cluster records that share the same key. Pool records keep back-references to their indexes to avoid hash re-computation and additional lookups during update and delete operations.

Our compiler analyzes the following access patterns: 1) scan over the entire collection (`foreach`), 2) lookup for a given unique key (`get`, `update`, `delete`), and 3) index scan for a given non-unique key (`slice`). We build a unique hash index for `get` operations, that is, when all lookup keys are bound at evaluation time. The same rule applies to `update`, `insert`, and `delete` operations. For `slice` operations, we create a non-unique index over the variables bound at evaluation time. When the access pattern analysis detects only `foreach` operations, we omit index creation.

DBToaster creates all relevant index structures. From our experience, most data structures produced during recursive compilation have only few indexes. For instance, the materialized

[2] Studying other index types, like B++ trees (for range operations) or binary heaps (for min/max), we leave for future work.

views produced compiling the TPC-H queries usually have zero or one secondary indexes, with rare exceptions of up to three non-unique indexes. Our empirical results indicate that the benefit of creating these indexes greatly outperforms their maintenance overheads.

Column-oriented layout

Record pools keep tuples in a row-oriented format. Materialized views store aggregated results in which all unused attributes are projected away during query compilation. Thus, during query evaluation, each access to one record likely references all its fields.

The row-oriented layout is, however, unsuitable for efficient data serialization and deserialization, which are important considerations in distributed query evaluation. To speed up these operations, the compiler creates array-based data structures for storing serializable data in columnar mode. It also generates specialized transformers for switching between row- and column-oriented formats.

We also use columnar data structures for storing input batches. Batched delta processing often starts by filtering out tuples that do not match query's static conditions. As these conditions are often simple (e.g., $A > 2$), using a columnar representation can improve cache locality. After filtering, we typically aggregate input batches to remove unused columns and store the result in a record pool.

4.3 Experiment Setup and Methodology

We evaluate the performance of DBToaster for single-tuple and batch updates in local settings. We analyze the throughput and cache locality of C++ view maintenance programs generated using the DBToaster Release 2.2, rev. 3387, released on November 27th, 2015 [5]. We compare our compilation algorithm with a commercial DBMS with incremental view maintenance capabilities (DBX) and a stream processing system (SPY). Since these systems are not optimized for our workload, we also provide a shared-infrastructure comparison by emulating their functionalities — query re-evaluation and IVM — within DBToaster-generated binaries. For batch experiments, we also compare DBToaster's recursive approach with re-evaluation and incremental computation in a PostgreSQL 9.4.5 database.

Experimental setup We run single-node experiments on an Intel Xeon E5-2630L @ 2.40GHz server with 2×6 cores, each with 2 hardware threads, 15MB of cache, 256GB of DDR3 RAM, and Ubuntu 14.04.2 LTS. We compile generated C++ programs using GCC 4.8.4. We estimate the memory consumption of our programs from the statistics returned by the `mallinfo` function.

We run experiments with a one-hour timeout on query execution, not counting loading of streams into memory and forming input batches of a given size.

4.3.1 Query and Data Workload

Our workload covers algorithmic order book trading (financial) and online business decision support scenarios TPC-H and TPC-DS [17, 16]. Figure 3.2 lists the query and evaluation properties of our workload.[3] Due to limitations of DBTOASTER, we made several changes to the TPC-H queries: (1) We ignored ORDER BY clauses and, to make the results comparable, also dropped the LIMIT clause; (2) We rewrote MIN, MAX aggregates using equivalent nested subqueries; (3) We replaced Q13's LEFT OUTER join with a natural join; (4) Finally, for convenience we rewrote HAVING clauses into subqueries and inlined INTERVAL expressions into constants. In experiments with batch processing, we also use a subset of TPC-DS queries from [97] (excluding four queries with the OVER clause, which we currently do not support).

The financial queries VWAP, MST, AXF, BSP, PSP, and BSV were run on a 2.63 million tuple trace of an order book update stream, representing one day of stock market activity for MSFT. The stream consists of updates to a Bids and Asks table with the schema (`timestamp, order_id, broker_id, price, volume`). We run the TPC-H and TPC-DS benchmark queries over data streams synthesized from TPC-H and TPC-DS databases by interleaving insertions to the base relations in a round-robin fashion. The default stream size is 10GB, unless stated otherwise.

4.3.2 DBToaster Setup

The DBTOASTER compiler produces incremental view maintenance code for both C++ and Scala. The compilers for these languages produce binaries with distinct (and surprising) performance characteristics [95]. In this thesis, our evaluation includes only C++ results.

DBTOASTER emulates the behavior of a traditional view maintenance system by terminating recursive delta materialization early. The remaining compiler stages (functional optimization and target-language generation) operate as usual. Our evaluation includes: (1) The HO-IVM algorithm as presented in this thesis (DBTOASTER), (2) A full re-evaluation of the query on every change (REEVAL), and (3) The HO-IVM algorithm used without recursion (first-order deltas are materialized) to emulate traditional IVM (IVM).

For each compilation method, we measured the memory consumption of the C++ programs. To this end, we produced instrumented binaries for each experiment and processed the same fraction of the stream as without profiling.

4.3.3 DBMS Setup

We compare DBTOASTER against a commercial DBMS. Due to licensing restrictions, we refer to it using the anonymized name DBX. In order to measure the rate at which DBX is able to refresh the query results as consistently as possible with other systems, we preload all updates to be performed on all base tables into a single table called Agenda. The Agenda table's schema

[3] The detailed queries can be found in Appendix A.1.

is the union of all of the input table schemas, and includes columns identifying the type of update (insert or delete), the table being updated, and the update's sequence number. Each trial iterates over the updates in Agenda in order, inserting or deleting one tuple and then refreshing the query results, either by re-evaluating the query (DBX-REEVAL), or by using the system's built-in capability to incrementally maintain materialized views (DBX-IVM). In order to minimize the overheads of the system, we disable log collection as much as possible.

For re-evaluation, we completely re-evaluate the query after each update and store its results in a separate table that gets truncated before each re-evaluation. Because generating materialized views that can be incrementally maintained is non-trivial, has many restrictions, and requires extra update logs, for IVM we use the provided tuning advisor in order to derive the proper view setup for each of the queries.

In many cases, the tuning advisor suggested views that were not precisely identical to the input queries. We encountered situations in which the advisor added group-by columns or relaxed WHERE clauses by dropping conditions or replacing disjunctions with single expressions, covering a superset of the original condition. We speculate that these transformations were meant to allow the generated view to support answering a larger class of queries. For complex queries that could not be maintained as a single view, the advisor generated nested subviews to be incrementally maintained and a top-level view to be re-evaluated on every commit.

4.3.4 SPY Setup

As a second comparison point, we use a commercial stream processor. We refer to the stream processor using the anonymized name SPY, again due to licensing restrictions. One major semantic difference between traditional stream processing engines and DBToaster is that stream processing engines are optimized to operate on windows of input streams, while DBToaster is designed to handle the whole history of a stream. We benchmark SPY by reading the same Agenda table used for DBX directly into a stream to minimize event dispatch overheads.

We implemented the queries using the dialect of SQL supported by SPY. Since the queries in our benchmark cannot be efficiently expressed using window semantics, we used auxiliary in-memory tables for all relations. Our implementation of the queries assigns a monotonically increasing number to each event and dispatches it to a stream corresponding to the affected relation. This stream updates the in-memory relation by inserting or removing the affected tuple. Then, the query result is re-evaluated and recorded together with the event number and a timestamp. Full recomputation is necessary as SPY does not support IVM.

Although we attempted to maintain the original query semantics, the SQL dialect employed by SPY imposes some limitations. A severe limitation is that in-memory tables may not be joined together; each in-memory table may only be joined with a stream, requiring manual selection of a join order. Our heuristic for this order was to minimize the size of intermediate streams.

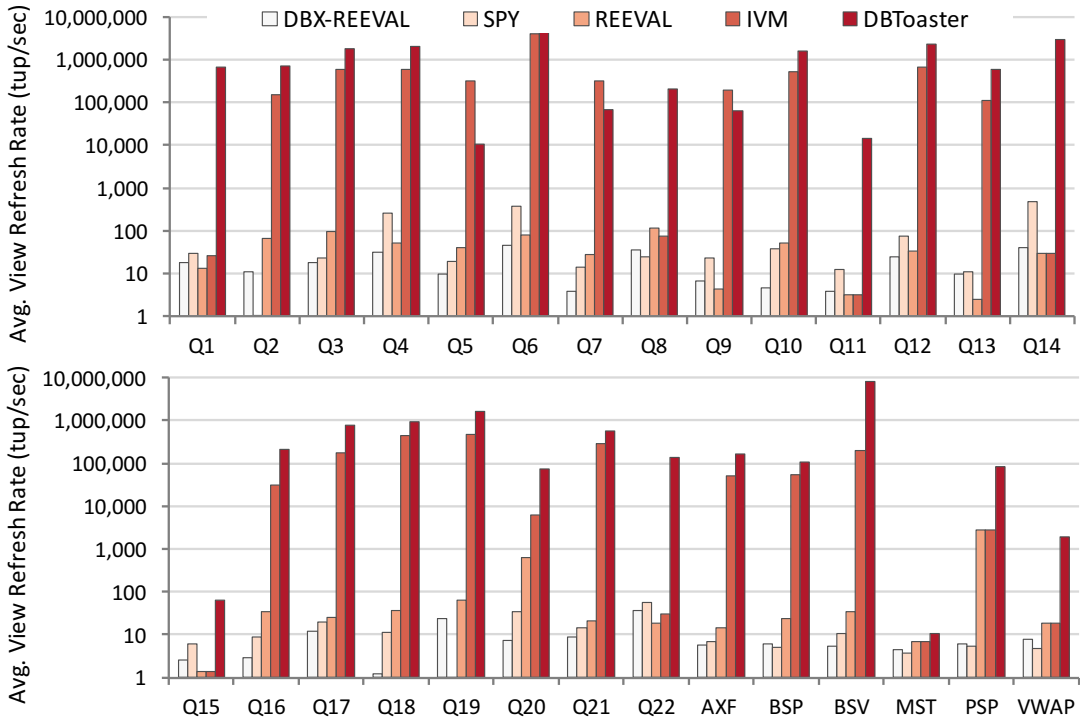


Figure 4.2 – DBTOASTER performance overview. Note the log scale on the y-axis.

4.4 Experiments with Single-tuple Updates

Our results show view refresh rates for single-tuple updates coming from the stream traces described in Section 4.3. These results show that:

- DBTOASTER consistently outperforms the two commercial systems we tested against by multiple orders of magnitude (Figures 4.2 and 4.3).
- DBTOASTER's Higher-Order IVM outperforms traditional IVM in all but three cases, and the speedup in roughly one-half of the workload varies from 2x to 7x and in several cases goes above 1,000x.
- DBTOASTER exhibits consistent performance over time while its memory requirements are often lower than with traditional IVM (Figures 4.4 and A.2).
- These results scale to longer streams with a fixed working set size (Figure 4.5).

In all figures, we use the following notation:

- **DBTOASTER** is the full HO-IVM algorithm.
- **REEVAL** and **IVM** are DBTOASTER repeatedly re-evaluating queries and emulating non-recursive IVM, respectively. **IVM PROJECT** improves **IVM** such that it retains in materialized views only the columns used in a given query.

	Query	DBX-REEVAL	DBX-IVM	SPY	REEVAL	IVM	DBTOASTER
TPC-H	Q1	17.7	1.6	29.5	13.4	25.4	681,073.3
	Q2	10.7	1.2	1.0	67.5	150,579.1	702,076.3
	Q3	18.2	0.7	22.6	95.0	576,024.9	1,740,859.8
	Q4	32.3	2.1	252.0	50.3	604,201.5	2,005,422.6
	Q5	9.7	0.3	19.1	39.3	319,409.4	10,321.8
	Q6	44.2	0.4	362.0	79.8	4,085,311.8	4,400,610.8
	Q7	3.8	1.4	14.1	28.1	307,225.1	68,422.9
	Q8	34.8	2.5	23.9	112.3	74.5	211,766.7
	Q9	6.8	1.8	23.7	4.4	194,584.2	64,244.4
	Q10	4.6	0.5	38.2	51.9	519,102.4	1,627,325.3
	Q11	3.7	2.8	12.3	3.3	3.3	14,374.5
	Q12	24.7	1.6	74.5	34.1	682,959.6	2,305,073.5
	Q13	9.6	2.9	10.9	2.5	109,936.3	605,765.6
	Q14	39.6	1.6	464.9	29.3	29.0	2,868,841.9
	Q15	2.5	1.9	6.1	1.4	1.4	62.7
	Q16	2.9	1.9	8.8	34.0	30,571.1	209,342.9
	Q17	11.8	2.1	19.6	25.6	173,673.4	773,782.3
	Q18	1.2	1.2	11.2	36.5	442,689.5	951,188.7
	Q19	23.8	1.4	0.6	65.3	463,468.1	1,655,056.2
	Q20	7.2	1.2	33.6	622.0	6,124.0	74,372.9
	Q21	8.7	1.3	14.7	20.6	297,190.3	568,108.7
	Q22	36.1	1.6	58.2	18.2	31.0	135,914.4
Finance	AXF	5.6	1.3	6.9	14.4	52,344.3	160,656.6
	BSP	6.0	1.6	5.2	24.1	54,920.8	108,354.0
	BSV	5.2	1.6	10.6	33.9	201,341.4	7,997,516.0
	MST	4.4	1.3	3.7	7.0	7.0	10.3
	PSP	5.9	2.0	5.4	2,741.2	2,743.1	85,086.6
	VWAP	7.9	2.1	4.8	18.1	18.4	1,934.8

Figure 4.3 – Comparison between DBTOASTER and two commercial query engines (in view refreshes per second). Both the DBMS (DBX) and stream system (SPY) columns show the cost of full refresh on each update. Higher numbers are better.

- **DBX-REEVAL** and **DBX-IVM** are a commercial database system performing view maintenance by re-evaluation and non-recursive IVM, respectively.
- **SPY** is a commercial stream processing engine.

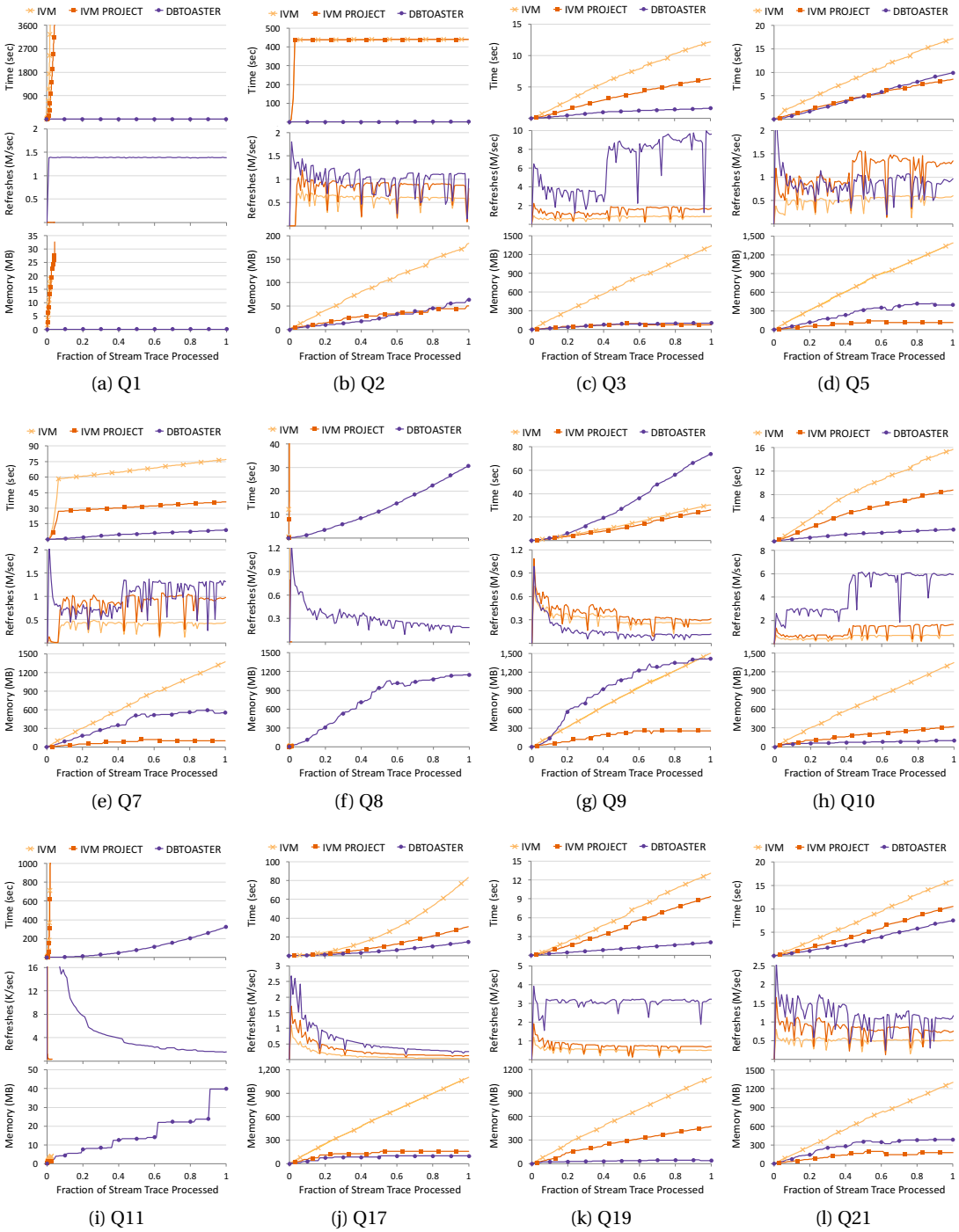
We benchmark DBTOASTER's C++ incremental programs. For comparison with Scala programs as well as the results of a naïve viewlet transform, which aggressively materializes entire delta queries and has few optimizations, we refer the reader to our paper [95].

4.4.1 Higher-Order IVM Performance

We now compare the performance of DBTOASTER with a commercial DBMS (DBX) and a stream processor (SPY).

Comparison with Commercial Systems

Figure 4.3 shows the performance of DBTOASTER's Higher-Order IVM alongside all comparison systems. We summarize our findings next.



(a) Q1 (b) Q2 (c) Q3 (d) Q5

(e) Q7 (f) Q8 (g) Q9 (h) Q10

(i) Q11 (j) Q17 (k) Q19 (l) Q21

Figure 4.4 – (a) Join-free query. (b) 5-way join with an equality/inequality correlated nested aggregate in the EXISTS clause. (c) 3-way join. (d) 6-way join. (e) 6-way join. (f) 8-way join. (g) 6-way star join. (h) 4-way join. (i) 2-way join with an aggregate subquery in the FROM clause and an uncorrelated nested aggregate. (j) 2-way join with an equality-correlated nested aggregate. (k) 2-way join with three disjunctive clauses. (l) 4-way join with an equality- and an inequality-correlated subqueries.

When recomputing the query results after each update (DBX-REEVAL), DBX experienced view refresh rates between 0.08 and 972.22 refreshes per second, with average and median values of 37.03 and 6. When using DBX's support for IVM, however, view refresh rates dropped to between 0.14 and 2.94, with average and median values of 1.42 and 1.38. This drop in performance when using IVM is counter-intuitive and prompted us to trace the execution of our program. DBX's tracing utility revealed that most of the execution time was spent parsing several parametrized system queries used in the bookkeeping. As the amount of useful work to be performed after a single update is quite small, the time spent parsing those system queries ends up dominating the overall running time. Maintaining catalog information across many tables for high rate updates also substantially impacts latencies and throughput.

The performance gap between SPY and DBToaster is a result of the lack of support for IVM in SPY and synchronization used to prevent the asynchronous system from producing inconsistent results. Due to the nature of the test queries, we are unable to make use of SPY's window semantics and are forced to use in-memory tables instead. Even though we use indexes on the in-memory tables wherever it makes sense, SPY seems to be unable to take full advantage of them in queries with complex predicates, contributing to poor performance, as exemplified in Q19.

Join-free Queries

The simplest queries in our workload, Q1 (Figure 4.4a) and Q6 (Figure A.3b), aggregate TPC-H's Lineitem relation. As these queries involve only one relation, the first-order delta depends solely on the inserted values.

The materialized view of Q6 stores a single aggregate value and has a constant update cost. Thus, the view refresh rates of the DBToaster and IVM methods are almost identical. The REEVAL compilation exhibits low refresh rates as it performs a complete scan over Lineitem upon every update. Unlike the other methods for which the memory overhead is negligible, REEVAL requires a linear size of memory to store all input tuples.

Q1 evaluates multiple group-by aggregates over Lineitem. DBToaster treats these aggregates as separate AGCA expressions and maintains each individually. Because the result set contains a fixed number of tuples (based on the limited domain of the group-by columns), DBToaster uses only a fixed amount of memory to store the additional maps.

DBToaster inlines the computation of algebraic aggregates. For instance, DBToaster computes averages from separate sum and count aggregates: Because the current incarnation of AGCA supports only one "multiplicity" per tuple, average is expressed as the product of the sum and inverse count. HO-IVM requires two recursive steps to separate out the (linear) count from the (non-linear) inverse count. This accounts for IVM's poor performance on Q1, as it must fully recompute the inverse count on every change. As future work, we plan to extend AGCA to generalize GMRs to have multiple "multiplicities". This will allow DBToaster to store multiple aggregate values per tuple, and improve the efficiency of this class of queries.

Equijoins

Q12 (Figure A.3c), Q14 (Figure A.3e), and Q19 (Figure 4.4k) contain two-way joins without nested aggregates. The first level deltas correspond nearly to the base relations. For Q12 and Q19, DBToaster materializes only the relevant columns of the base relations for the tuples matching query predicates. This strategy yields lower maintenance costs and memory requirements than both IVM and IVM PROJECT. As in Q1, Q14 has to maintain an inverse count, resulting in poor performance for IVM. The domain extraction procedure (Section 3.4.2) in the deltas of nested aggregates of these three queries allow us to avoid a full scan of each materialized nested aggregate whenever it changes.

Query decomposition also plays an important role in efficiency of DBToaster for queries containing linear joins of 3 or more relations. Decomposition avoids materialization of cross products, improving performance and reducing memory consumption. For instance, the delta of Q10 (a 4-way equijoin) with respect to the Orders relation creates a cross product between Customer and Lineitem (which are only connected through Orders in the original query).

Due to the foreign key constraints in the TPC-H schema (which DBToaster is not aware of) most loops in Q3's trigger program have only one iteration, and the cost of updating either the Orders or Lineitem relation is constant. For queries with multi-way joins and selection predicates – Q3 (Figure 4.4c), Q5 (Figure 4.4d), and Q10 (Figure 4.4h)) – DBToaster further outperforms IVM and IVM with projection by pushing predicates into the materialized views and dropping unused columns.

DBToaster considers the contents of Nation and Region relations as static, which it loads into memory before processing the streams. It avoids materialization of deltas needed to support updates to these relations, effectively reducing the join width of certain queries (Q5 and Q10) and eliminating several potentially high maintenance maps.

Nested Aggregates

Q17 (Figure 4.4j) is a multi-way join query with nested aggregates that are correlated on an equality with the outer query. In this case, both DBToaster and IVM benefit from decorrelating the nested subquery and range-restricting the domain of the generated delta expressions for updates to the Lineitem relation (on which all nested subqueries are based).

VWAP (Figure 4.3) has a nested aggregate correlated on an inequality. The small domain of the correlation variable (`price`) makes this an ideal candidate for view caching [95].

PSP (Figure 4.3) includes two uncorrelated nested aggregates. It benefits from top-level query re-evaluation on each update. As in Section 3.3.2, polynomial expansion and graph decomposition are essential to avoid computation of a cross product between the base relations. DBToaster evaluates the query using six auxiliary materialized views with constant time updates: Two views maintain single aggregate values, while the others are linear in the number

of distinct values of the column being compared to the nested aggregate (`volume`). The finite domain of these values results in a nearly constant refresh rate and memory consumption.

MST (Figure 4.3) is fundamentally similar to PSP, but rather than comparing its uncorrelated aggregates against columns from the base relations, they are each compared against another nested aggregate correlated on an inequality. This is a worst case scenario for DBToaster, as it cannot incrementally process this query in better than $O(n^2)$ time without specialized indexes (e.g., aggregate range trees).

Inequijoins

AXF (Figure 4.3) and BSP (Figure 4.3) are 2-way joins with inequality join-predicates. The performance graph of AXF shows the inefficiency of view caching in this case. The view caching approach treats both the join variable (`price`) and one of the aggregate variables (`volume`) as input variables; together, these input variables have an extremely large domain. In BSP, the join variable (timestamp) also has an unbounded domain. In both cases, DBToaster outperforms view caching by precluding materialized views with input variables. DBToaster also achieves a small speed boost compared to IVM by not materializing the entire base relation.

Queries with EXIST or IN Clauses

Q2 (Figure 4.4b), Q4 (Figure A.3a), Q16 (Figure A.3g), and Q21 (Figure 4.4l) contain clauses that check for the existence of the nested subquery results. DBToaster transforms each subquery into a count aggregate, assigns this value to a fresh variable, and adds an additional constraint over that variable according to the semantics of the clause (e.g., $x = 0$ for the NOT EXIST clause). As all the subqueries of the above queries are correlated on an equality, DBToaster decides to incrementally maintain the top-level views for updates to the subquery relations. For queries that are also correlated on an inequality (Q2 and Q21), DBToaster avoids materializing maps with input variables due to the large domain of the correlation variables (`supplycost` and `suppkey`, respectively). Q21 has constant time updates to Lineitem and Orders, and a linear time update in the number of orders for one supplier. But since the number of suppliers in the stream is fairly small, the refresh rate remains roughly constant once the memory consumption stabilizes.

Subqueries in FROM Clauses

DBToaster maintains separate materialized views for subqueries that appear in the FROM clause (Q7, Q8, and Q9). For Q9 (Figure 4.4g), DBToaster materializes large intermediate views whose sizes grow with time, which yields lower refresh rates than both IVM techniques. For Q7 (Figure 4.4e), DBToaster's memory requirements are significant but remain constant after processing roughly one-half of the stream at which point DBToaster outperforms IVM with projection. The complexity of Q8 (Figure 4.4f) causes poor performance with both IVM techniques, which manage to process only a tiny fraction of the stream. DBToaster achieves better performance at the cost of increased memory consumption.

Complex Queries

The remaining TPC-H queries Q11, Q13, Q15, Q18, Q20, and Q22 combine the above characteristics. Our experiments show that the update costs for these queries coincide with their structural complexity.

Q11 (Figure 4.4i), has a group-by aggregate in its FROM clause and an uncorrelated nested aggregate that appears in an inequality at the top level. DBTOASTER exploits the fact that both subqueries share the same structure to reduce the number of generated maps. The costs of updating Supplier and Partsupp are linear in the number of distinct `partkey` values. Since the update stream contains only insertions to the base relations, the amount of memory used to store additional views grows continuously and, consequently, the view refresh rate continuously drops.

Q15 (Figure 4.4i) is a variation of the original TPC-H query where a nested subquery and an EXIST clause replace the max aggregate. Since both subqueries are identical, duplicate view elimination reduces the number of auxiliary views. However, the update cost for this query grows quadratically with the number of distinct `suppkey` values in Lineitem, as shown on the graph. DBTOASTER manages to compute only 1% of the input stream. To improve the performance of MIN, MAX, and theta-joins in general, we plan to extend DBTOASTER with specialized tree-based data structures.

4.4.2 Stream Scalability

This section analyzes the scaling behavior of DBTOASTER for a subset of the TPC-H queries over a larger stream of updates. Our focus is on measuring view refresh rates in terms of the stream length and query complexity, rather than the working set size.

The workload for this experiment was synthesized from databases created by DBGEN at scaling factors 0.1, 0.5, 1, 5, and 10 (100MB, 500MB, 1GB, 5GB, and 10GB, respectively). An update stream was built by randomly interleaving tuples from the base relations, while preserving the reference integrity. After inserting 30,000 Orders tuples and around 120,000 Lineitem tuples, we randomly inject deletions into these two relations in order to keep their sizes roughly constant. Tuples in other TPC-H relations are never deleted. All updates preserve the foreign key constraints that exist between the TPC-H tables.

The length of the update stream increases with larger scaling factors. However, the size of the working set depends on the query structure. Materialized views that reference Customer, Part, Supplier, or Partsupp might grow with larger scaling factors, while views defined solely over Orders or Lineitem have a bounded working set size.

Figure 4.5 presents the results of our scaling experiments. For most queries performance stays roughly constant as the stream length grows. Q2 and Q16 select over insert-only relations (Part, Supplier, and Partsupp); thus, the memory overhead of DBTOASTER grows with the scaling

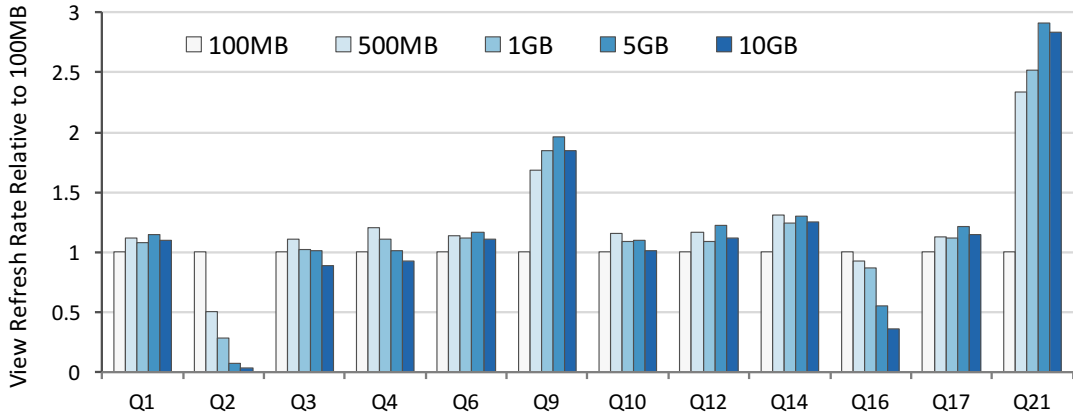


Figure 4.5 – Performance scaling on a subset of TPC-H queries.

factors. The view refresh rates drop as the maintenance cost for these queries is linear in the size of in-memory data structures. Q9 and Q21 demonstrate an increase of the view refresh rates for larger stream lengths. The reason for this behavior is as follows. In our workload, the working set sizes of Orders and Lineitem are constant, regardless of the scaling factor. With larger scaling factors the base relations get larger; thus, we have to place more deletions to maintain the size invariant (As an extreme case, imagine that the working set size of Orders is 1; then we have to double the number of Orders tuples in the stream as every insertion is followed by a deletion). Placing more deletions increases the fraction of Orders and Lineitem tuples in the stream. This in turn affects the view refresh rates of these queries, as both have constant costs with respect to updates to the Lineitem relation.

4.4.3 Memory requirements

Higher-order incremental view maintenance materializes auxiliary views to speed up the work required to keep all views fresh. The sizes of these auxiliary views created to maintain a given query, in general, depend on the query structure. Our query workloads, TPC-H and TPC-DS, are based on the star schema with one large fact table and several dimension tables. In such cases, auxiliary materialized views cannot have more tuples than the fact table due to integrity constraints. In practice, the sizes of materialized views are much smaller than the sizes of the fact tables because: (1) their view definition queries often involve static predicate, and (2) the views discard columns unused in a given query and aggregate over the remaining columns.

In our experiments, both incremental view maintenance strategies store only base relations for the purpose of computing delta queries. Traditional IVM materializes entire base relations, while IVM PROJECT stores only the columns used in a given query (but ignores query predicates). From the memory graphs in Figure 4.4 and A.2, we observe that the total memory consumption in DBToaster is almost always smaller than in IVM and, in many cases, IVM PROJECT. The latter is a consequence of pushing query predicates into materialized views.

4.5 Experiments with Batch Updates

We evaluate the performance of recursive incremental view maintenance for batch updates of different sizes in local settings. We analyze the throughput and cache locality of DBToaster C++ incremental programs. Our experimental results show that:

- In local mode, view maintenance code specialized for tuple-at-a-time processing can outperform or be on par with batched programs for almost half of our queries.
- Preprocessing input batches can boost the performance of incremental computation by multiple orders of magnitude.
- Large batches can have negative impacts on cache locality. In local mode, the throughput of most of our queries peaks for batches with $1,000 – 10,000$ tuples.

Our local experiments compare tuple-at-a-time and batched incremental programs. The former have triggers with tuple fields as function parameters, which can be inlined into delta computation; the latter consist of triggers accepting one arbitrary-sized columnar batch. In both cases, we generate single-threaded C++ code.

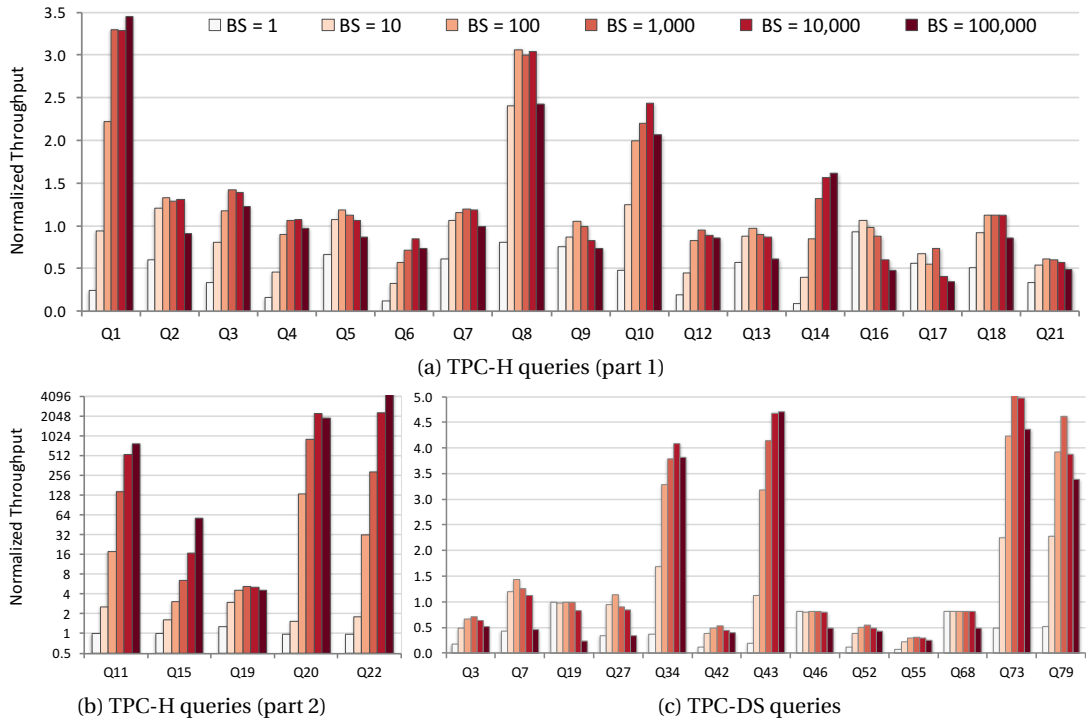
Batched incremental programs have extra loops inside triggers for processing input batches. To avoid redundant iterations over the whole batch, we materialize input tuples that match query's static conditions, retaining only the attributes (columns) used in incremental evaluation. These pre-aggregated batches are smaller in size due to having fewer attributes and, potentially, fewer matching tuples, which can significantly affect view maintenance costs. In contrast, single-tuple triggers avoid materialization of input batches.

4.5.1 Batch Size vs. Throughput

Figure 4.6 shows the normalized throughput of batched incremental processing of the TPC-H queries for different batch sizes using the tuple-at-a-time performance as the baseline. For ease of presentation, we use two graphs with different y-axis scales.

For almost half of our queries, batched incremental processing performs worse or just marginally better than specialized tuple-at-a-time processing. This result comes at no surprise once we start analyzing the pre-aggregated batches of these queries for updates to their largest (and usually most expensive) relation. For instance, Q5, Q9, and Q18 aggregate Lineitem deltas by `orderkey` and, possibly, some other fields; Q16 aggregates Partsupp deltas by the primary key (`partkey`, `suppkey`). In these cases, batch pre-aggregation retains (almost) the same number of input tuples, bringing no performance improvements; on the contrary, it introduces extra materialization and looping overheads. For two-way join queries, Q4, Q12, and Q13, the simplicity of their view maintenance code makes these overheads particularly pronounced.

Batch pre-aggregation keeps only input tuples that match query predicates. This step can accelerate the rest of view maintenance code by a factor that depends on the selectivity of



(a) TPC-H queries (part 1)

(b) TPC-H queries (part 2)

(c) TPC-DS queries

Figure 4.6 – Normalized throughput of TPC-H and TPC-DS queries for different batch sizes with single-tuple execution as the baseline.

predicates. For instance, preprocessing in Q3, Q7, Q8, Q10, and Q14 filters out tuples of Lineitem and Orders batches and yields improvements from 19% in Q7 to 306% in Q8.

Batch pre-aggregation can project input tuples onto a set of attributes with small active domains. For instance, Q1 projects a Lineitem batch onto columns containing only few possible values, which enables cheap maintenance of the final 8 aggregates. So, the single-tuple implementation performs more maintenance work per input tuple. For Q2 and Q19, batch filtering and projection can give up to 1.3x and 5.1x better performance. This benefit can increase for queries with more complex trigger functions. For instance, Q22 filters and projects an Orders batch on `custkey`, while Q20 filters and projects Partsupp and Lineitem batches on `suppkey`. In both cases, the projected columns have much smaller domains, bringing significant improvements, 2,243x in Q20 and 4,319x in Q22.

Incremental programs for single-tuples updates are easier to optimize than their batched counterparts due to having simpler input and fewer loops in trigger bodies. This virtue emerges in Q17 and Q21. For these queries, our compiler fails to factorize common subexpressions as efficiently as during the single-tuple compilation, which causes same expressions to be evaluated twice.

For Q11 and Q15, incremental view maintenance is more expensive than re-evaluation due to inequality-based nested aggregates. Increasing batch sizes results in fewer re-evaluations, which increases the overall throughput at the expense of higher latencies.

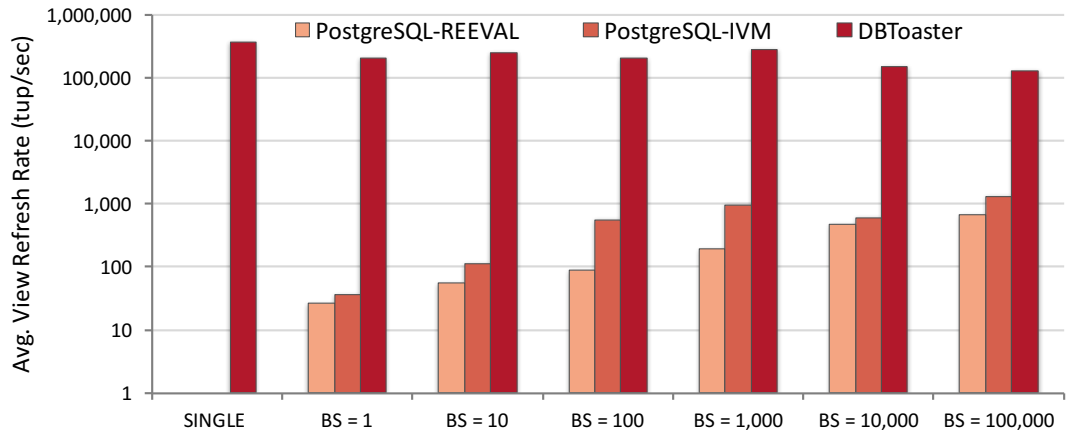


Figure 4.7 – Throughput comparison for TPC-H Q17 of re-evaluation and incremental maintenance in PostgreSQL and recursive incremental maintenance in generated C++ code for different batch sizes. SINGLE denotes specialized single-tuple processing in C++.

Bulk processing amortizes the overhead of invoking trigger functions. For instance, maintaining a single aggregate over Lineitem in Q6 with batches of 10,000 tuples can bring up to 2x better performance than the single-tuple execution. However, these results hold only when function inlining is disabled. Since the single-tuple trigger body of Q6 consists of just one conditional statement, the C++ compiler usually decides to inline this computation, causing single-tuple execution to always outperform batched execution.

Figure 4.6c shows the normalized throughput of batched incremental processing of a subset of the TPC-DS queries for different batch sizes using the tuple-at-a-time performance as the baseline. These results show that single-tuple processing of TPC-DS queries often outperforms batched processing due to simpler maintenance code. Preprocessing input batches to filter out irrelevant tuples and remove unused columns can bring up to 5x better performance for four TPC-DS queries from our workload.

Comparison with PostgreSQL

Figure 4.7 compares the throughput of recursive incremental processing in C++ and re-evaluation and classical incremental view maintenance in PostgreSQL for TPC-H Q17. We implement incremental processing in PostgreSQL using the domain extraction procedure described in Section 3.4. The results show that the generated code outperforms PostgreSQL re-evaluation from 233x to 14,181x and classical incremental view maintenance from 120x to 10,659x, for different batch sizes.

Appendix A.4 contains the performance numbers for the TPC-H and TPC-DS queries from our workload. These numbers demonstrate that our view maintenance and code specialization techniques outperform, in all but four cases, the database system by orders of magnitude, even when processing large update batches.

Batch size	Single	1	10	100	1000	10000	100000
Instructions	19,633	145,670	33,407	17,199	15,750	15,425	15,868
I1 misses	2.0	6.8	3.5	2.0	1.8	1.8	1.4
LLC references	485	683	533	424	402	578	668
LLC misses	369	562	416	302	258	302	316

Table 4.1 – Cache locality of TPC-H Q3. All numbers are in millions.

4.5.2 Cache Locality

In this experiment, we measure the cache locality of generated programs for single-tuple and batched incremental processing on a 10GB input stream. We use perf 3.13.11 to monitor CPU performance counters during the view maintenance of TPC-H Q3. We start profiling after loading the streams and forming input batches. Table 4.1 presents the obtained results.

The numbers of retired instructions of the single-tuple and batched programs roughly correspond to the normalized throughput numbers from Figure 4.6. The batch processing with size 1 executes almost 10x more instructions than with size 1,000, which translates into a 4.3x slowdown in running time. The generated programs exhibit low numbers of L1 instruction cache misses compared to the total number of instructions. These results are indicators of good code locality of our view maintenance code.

The second block shows the number of caches references and misses that reached the last level cache. Extremely large and small batch sizes have negative effects on cache locality. Relatively high numbers of cache misses are expected given that in this streaming scenario most input data passes through the query engine clearing out the cache and without being referenced again. Batch processing with size 1,000 exhibits the lowest number of cache references and cache misses, which corresponds to the result from Figure 4.6.

4.6 Summary

The DBToaster system implements Higher-Order IVM using modern code and data structure generation techniques. DBToaster outperforms commercial data management systems on a workload consisting of financial and decision support queries by multiple orders of magnitude. We also study the effect of batch size on the latency of processing in local settings: we show that pre-processing input batches can boost the performance of incremental computation but also demonstrate that tuple-at-a-time processing can outperform batch processing using code specialization techniques (in roughly one-half of the benchmarked queries).

5 Distributed Incremental View Maintenance

Emerging data-intensive applications demand scalable processing to speed up query evaluation and accommodate growing memory and storage requirements. In this section, we extend DBToaster with techniques that transform maintenance triggers generated for local execution into data-parallel programs optimized for running on large-scale processing platforms with synchronous execution, like Spark [152, 151] or Hadoop [1]. Our approach is general and applies to any input program formed using our query language, not just recursive view maintenance programs.

Distributed Execution Model We assume a synchronous execution model where one driver node orchestrates job execution among workers, like in Spark or Hadoop. Processing one batch of updates may require several computation stages, where each stage runs view maintenance code in parallel. All workers are stateful, preserve data between stages, and participate in every processing stage.

Our approach naturally leverages the fault tolerance mechanisms of the underlying execution platform, in our case, the Spark computing framework. Using data checkpointing, we can periodically save intermediate state to reliable storage (HDFS) to shorten recovery time. Checkpointing may have detrimental effects on the latency of processing, so the user needs to carefully tune the frequency of checkpointing based on application requirements.

Types of Materialized Views We classify materialized views depending on the location of their contents. *Local views* are stored and maintained entirely on the driver node. They are suitable for materializing top-level aggregates with small output domains. *Distributed views* have their contents spread over all workers to balance CPU and memory pressure. Each distributed view has an associated partitioning function that maps a tuple to a non-empty set of nodes storing its replicas.

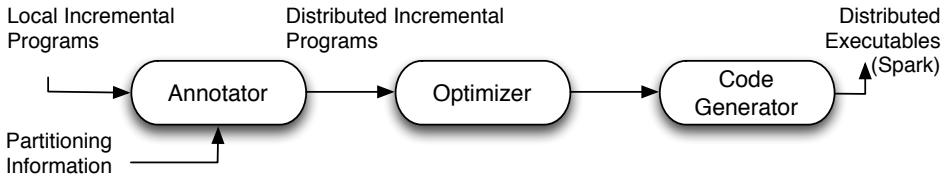


Figure 5.1 – Compilation of AGCA incremental programs

5.1 Compilation Overview

Figure 5.1 shows the process of transforming a local incremental program into a functionally equivalent, distributed program. The input consists of statements expressed using our query language. To distribute their execution, the compiler relies on partitioning information about each materialized view. Deciding on the optimal view partitioning scheme happens outside of the compilation process; we assume that such partitioning information is available (e.g., provided by the user).

The compilation process consists of three phases. First, we annotate a given input program with partitioning information and, if necessary, introduce operators for exchanging data among distributed nodes in order to preserve the correctness of query evaluation. Next, we optimize the annotated program using a set of simplification rules and heuristics that aim to minimize the number of jobs necessary for processing one update batch. Finally, we generate executable code for a specific processing platform. Only the code generation phase depends on the target platform.

5.2 Well-formed Distributed Programs

The semantics of the query operators, presented in Section 2.2.2, cannot be directly translated to distributed environments. For instance, unioning one local and one distributed view has no clear meaning. Even among views of the same type, naïvely executing query operators at each distributed node might yield wrong results. For instance, a natural join between two distributed views produces a correct result only if the views are identically partitioned over the join (common) keys; otherwise, one or both operands need to be repartitioned.

We extend our query language with new location-aware primitives that allow us to construct *well-formed* query expressions. Such expressions preserve the correctness of distributed query evaluation for the given partitioning strategy.

Location Tags To reason about the semantics and correctness of query evaluation, we annotate query expressions with location tags: (1) `Local` tag denotes the result is located on the driver node; (2) `Dist(`$\mathcal{P}$`)` tag marks the result is distributed among all workers according to partitioning function $\mathcal{P}$; and 3) `Random` tag denotes the result is randomly distributed among all workers.

A relation (materialized view) can take on a `Local`, `Dist`, or `Random` tag. Constants, values, comparisons, and variable assignments involving values are interpreted (virtual) relations whose contents is deterministic and never materialized. The location of such terms is irrelevant from the perspective of query evaluation, and they can freely participate in both local and distributed expressions; in distributed settings, one can view these terms as fully replicated.

Location Transformers To support distributed execution, we extend the language with new operators for manipulating location tags and exchanging data over the network.

- $\texttt{Repart}_{\mathcal{P}_2}(Q^{\{\texttt{Dist}(\mathcal{P}_1),\texttt{Random}\}}) = Q^{\texttt{Dist}(\mathcal{P}_2)}$
  Partition the distributed result of Q using function $\mathcal{P}_2$.
- $\texttt{Scatter}_{\mathcal{P}}(Q^{\texttt{Local}}) = Q^{\texttt{Dist}(\mathcal{P})}$
  Partition the local result of Q using function $\mathcal{P}$.
- $\texttt{Gather}(Q^{\{\texttt{Dist}(\mathcal{P}),\texttt{Random}\}}) = Q^{\texttt{Local}}$
  Aggregate the distributed result of Q on the driver node.

The location transformers are the only mechanism for exchanging data in our distributed programs. `Repart` and `Gather` operate over distributed expressions, while `Scatter` supports only local expressions.

Distributed Query Operators We extend the semantics of our AGCA operators with location tags as follows.

- Relation $R(A_1, A_2, ...)^T$ stores the contents at location T.
- Bag union $\alpha^T + \beta^T$ merges tuples either locally or in parallel on every node, and the result retains tag T. Requires the same location tag for both operands.
- Natural join $\alpha^T * \beta^T$ has the usual semantics when $T = \texttt{Local}$. For distributed evaluation, both operands need to be partitioned on the join keys; the result is distributed and consists of locally evaluated joins on every node. Joins on `Random` are disallowed.
- $\text{Sum}_{[A_1,A_2,...]}Q^T$, when $T = \texttt{Dist}(\mathcal{P})$, computes partial aggregates on every node. The result has tag T only if Q is key partitioned on one of the group-by columns; otherwise, we annotate with a `Random` tag. In other cases, the result retains tag T.

All other language constructs – constants, values, comparisons, and variable assignments – are location independent and their semantics remain unchanged.

Next, we present an algorithm for transforming a local program into a well-formed distributed program based on the given partitioning information. The algorithm annotates and possibly extends the expression trees of the statements to preserve the semantics of each operator. For each statement, we start by assigning location tags to all relational terms in the expression tree. Then, in a bottom-up fashion, we annotate each node of the tree with a location tag. We

introduce location transformers where necessary to preserve the semantics and correctness of each query operator, as discussed above. Upon reaching the root node, we ensure that the RHS query expression evaluates at the same location where the target LHS view is materialized, and, if necessary, introduce a `Gather` or `Scatter` transformer. To obtain a well-formed distributed program, we apply this procedure to every input statement.

Example 5.2.1 Let us construct a well-formed statement for

$$M(A) \mathrel{+}= \mathrm{Sum}_{[A]}(M_1(A,B) * M_2(A,B))$$

when $M(A)$ and $M_1(A,B)$ are partitioned by A, while $M_2(A,B)$ is partitioned by B. We associate location tags to the materialized views. We use $M(A)^{[A]}$ to denote that $M(A)$ is partitioned on column A. Since the join operands have incompatible location tags, we introduce `Repart` around one of the operands (e.g., left):

$$\mathtt{Repart}_{[B]}(M_1(A,B)^{[A]})^{[B]} * M_2(A,B)^{[B]}$$

The join result remains partitioned on B. The Sum expression computes partial aggregates grouped by column A. Such an expression cannot have location tag $[B]$ since B is not in the output schema; so, we assign a `Random` tag to the expression. Now, we have reached the root of the expression tree. We need to ensure that the RHS expression has the same location tag as the target view. So, adding a `Repart` transformer produces a well-formed statement:

$$M(A)^{[A]} \mathrel{+}= \mathtt{Repart}_{[A]}(\mathrm{Sum}_{[A]}(\quad \mathtt{Repart}_{[B]}(M_1(A,B)^{[A]})^{[B]} * M_2(A,B)^{[B]})^{Random})^{[A]}$$

□

The algorithm for constructing well-formed expressions has no associated cost metrics and might produce suboptimal solutions. In the above example, executing the final statement requires two communication rounds (`Reparts`) between the driver and workers. Such communication overhead is unnecessary – if we repartition the other join operand, we produce an equivalent, less expensive well-formed statement with only one communication round:

$$M(A)^{[A]} \mathrel{+}= \mathrm{Sum}_{[A]}(M_1(A,B)^{[A]} * \mathtt{Repart}_{[A]}(M_2(A,B)^{[B]})^{[A]})^{[A]}$$

To optimize well-formed programs we rely on a set of simplification and heuristic rules.

5.3 Optimizing Distributed Programs

In this section, we describe how to optimize well-formed programs in order to minimize communication and processing costs. We divide this task into two stages. The first stage relies on a simple cost-based model to simplify each individual statement. The second stage exploits commonalities among statements to avoid redundant communication and minimize the number of jobs needed to execute the given program.

$$\texttt{Repart}_{\mathcal{P}}(\alpha * \beta) \Leftrightarrow \texttt{Repart}_{\mathcal{P}}(\alpha) * \texttt{Repart}_{\mathcal{P}}(\beta)$$

$$\texttt{Repart}_{\mathcal{P}}(\alpha + \beta) \Leftrightarrow \texttt{Repart}_{\mathcal{P}}(\alpha) + \texttt{Repart}_{\mathcal{P}}(\beta)$$

$$\texttt{Repart}_{\mathcal{P}}(\text{Sum}_{[A_1,A_2,\ldots]}(Q)) \Leftrightarrow \text{Sum}_{[A_1,A_2,\ldots]}(\texttt{Repart}_{\mathcal{P}}(Q))$$

$$\texttt{Repart}_{\mathcal{P}}(var := Q) \Leftrightarrow (var := \texttt{Repart}_{\mathcal{P}}(Q))$$

Figure 5.2 – Bidirectional optimization rules for location transformers. The same rules hold for `Repart`, `Scatter`, and `Gather`.

5.3.1 Intra-Statement Optimization

Intra-statement optimization aims to minimize the amount of network traffic required to execute one well-formed statement. Our cost model takes the number of communication rounds as the basic metric for cost comparison. Each location transformer (`Repart`, `Scatter`, and `Gather`) shuffles data over the network, so statements containing fewer of them require less communication. Our cost model also relies on few heuristics to resolve ties between statements with the same cost. We reshuffle expressions that involve batch updates rather than whole materialized views as the formers are usually smaller in size; we favor expressions with fewer `Gather` transformers to distribute computation as much as possible.

The optimizer uses a trial and error approach to recursively optimize a given statement. It tries to push each location transformer down the expression tree, following the rules from Figure 5.2. Note that these rules might produce more expensive expressions, in which case the algorithm backtracks. During each optimization step, the compiler tries to simplify the current expression using the rules from Figure 5.3. Each simplification rules always produces an equivalent expression with fewer location transformers.

In Example 5.2.1, the optimizer pushes the outer $\texttt{Repart}_{[A]}$ through Sum and $*$, and then simplifies $\texttt{Repart}_{[A]} \circ \texttt{Repart}_{[B]}$ to $\texttt{Repart}_{[A]}$. The optimized statement requires only one communication round.

5.3.2 Inter-Statement Optimization

Location transformers represent natural pipeline breakers in query evaluation since they need to materialize and shuffle their contents before continuing processing. In this section, we show how to analyze inter-statement dependencies to minimize the number of pipeline breakers and their communication overhead.

Single Transformer Form To facilitate inter-statement analysis, we first convert a given program into single transformer form where each statement has at most one location transformer. The transformer, if present, always references one materialized view. In other words, we normalize the input program by: (1) materializing the contents being transformed (if not already

$$\texttt{Repart}_{\mathcal{P}}(Q^{\texttt{Dist}(\mathcal{P})}) \Rightarrow Q^{\texttt{Dist}(\mathcal{P})} \qquad \texttt{Gather} \circ \texttt{Repart}_{\mathcal{P}} \Rightarrow \texttt{Gather}$$

$$\texttt{Gather}(Q^{\texttt{Local}}) \Rightarrow Q^{\texttt{Local}} \qquad \texttt{Gather} \circ \texttt{Scatter}_{\mathcal{P}} \Rightarrow \texttt{Gather}$$

$$\texttt{Repart}_{\mathcal{P}_1} \circ \texttt{Repart}_{\mathcal{P}_2} \Rightarrow \texttt{Repart}_{\mathcal{P}_1} \qquad \texttt{Scatter}_{\mathcal{P}} \circ \texttt{Gather} \Rightarrow \texttt{Repart}_{\mathcal{P}}$$

$$\texttt{Repart}_{\mathcal{P}_1} \circ \texttt{Scatter}_{\mathcal{P}_2} \Rightarrow \texttt{Scatter}_{\mathcal{P}_1}$$

Figure 5.3 – Simplification rules for location transformers. ∘ denotes operator composition.

materialized), (2) extracting the location transformers targeting the materialized contents into separate statements, and (3) updating the affected statements with new references. To achieve that, we recursively bottom-up traverse the expression tree of every statement.

Single transformer form sets clear boundaries around the contents that needs to be communicated, which eases the implementation of further optimizations. We apply common subexpression elimination and dead code elimination to detect expressions shared among statements and to eliminate redundant network transfers. In contrast to the classical compiler optimizations, our routines are aware of the location where each expression is executed.

Statement Execution Mode The location tag associated with each statement determines where and how that statement is going to be executed. Statements targeting local materialized views are, as expected, executed at the driver node in *local mode*. Statements involving distributed views run in *distributed mode*, where the driver initiates the computation.

All location transformations run in local mode since the driver governs their execution. But, materializing the contents to be reshuffled can happen in both local and distributed mode. For instance, preparing contents for `Scatter` takes place on the driver node, while for `Repart` and `Gather` happens on every worker node.

Statement Blocks Distributed statements are more expensive to execute than local statements. To run a distributed statement, the driver needs to serialize the task closure, ship it to all the workers, and wait for the completion of each one of them. For short-running tasks, and recursive view maintenance is often such, non-processing overheads can easily dominate in the execution time.

To amortize the cost of executing distributed statements, we pack them together into processing units called *statement blocks*. A statement block consists of a sequence of distributed statements that can be executed at once on every node without comprising the program correctness. Apart from distributed blocks, we also introduce blocks of local statements whose purpose is to determine which network operations can be batched together (the driver initiates `Repart`, `Scatter`, and `Gather` in local mode).

Data-flow dependencies among statements prevent arbitrary re-orderings of statements and blocks. Figure 5.4 shows methods for checking the commutativity of statements and blocks.

Listing 5.1 The block fusion algorithm

```
1  def commute(s1: Stmt, s2: Stmt): bool =
2    !s2.rhsMaps.contains(s1.lhsMap) &&
3    !s1.rhsMaps.contains(s2.lhsMap)
4
5  def commute(b1: Block, b2: Block): bool =
6    b1.stmts.forall(lhs =>
7      b2.stmts.forall(rhs => commute(lhs, rhs)))
8
9  def mergeIntoHead(hd: Block, tl: List[Block]) =
10   tl.foldLeft (hd,Nil) { case ((b1,rhs),b2) =>
11     if (b1.mode == b2.mode &&
12         rhs.forall(b => commute(b,b2)))
13       (Block(b1.mode, b1.stmts++b2.stmts), rhs)
14     else (b1, rhs :+ b2) }
15
16 def blockFusion(blocks: List[Block]) = blocks match {
17     case Nil => Nil
18     case hd::tl =>
19       val (hd2,tl2) = mergeIntoHead(hd,tl)
20       if (hd == hd2) hd::blockFusion(tl)
21       else blockFusion(hd2::tl2) }
```

Figure 5.4 – The block fusion algorithm

Block Fusion Algorithm We prefer program execution plans with as few statement blocks as possible. Here, we describe an algorithm that reorders and merges together consecutive blocks to minimize their number. Figure 5.4 presents the algorithm.

First, we promote each statement into a separate block that keeps statement's execution mode (local or distributed). Then, the algorithm tries to fuse together the first block with the others that share the same execution mode and commute with all intermediate blocks. On success, the algorithm merges the new block sequence recursively; otherwise, it handles the remaining blocks recursively.

Figure 5.5 visualizes the effects of block fusion on the incremental program of TPC-H Q3. Before running the algorithm, the annotated input program contained 10 local and 12 distributed statement blocks. After reordering and merging these blocks, the algorithm outputs only 2 local and 2 distributed compound blocks.

5.4 Code Generation

Statement blocks considerably simplify code generation. Isolating distributed blocks enables workers to safely run code generated for single-node execution on their local data partitions. Pure local statements without transformers also correspond to unmodified single-node code. Distributed code generation, thus, relies to a great extent on single-node code generation.

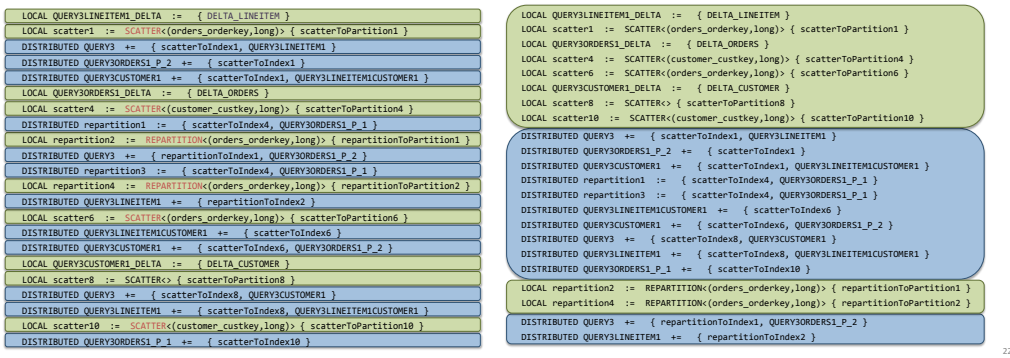


Figure 5.5 – The block fusion effect in TPC-H Q3: before and after. Green blocks are local, blue blocks are distributed. The initial program has 22 blocks (10+12), while the optimized program has 4 blocks (2+2).

Code generated for location transformers uses platform-specific communication primitives for exchanging data (e.g., in Spark, we use the shuffling operation, which is implicit in many RDD transformations). To minimize network overhead, we encapsulate transformers of the same type into one compound request per block. For instance, we coalesce multiple `Scatter` transformers and their materialized data into just one `Scatter` request that uses a container data structure.

5.5 Experimental Evaluation

We evaluate the performance of DBToaster's incremental view maintenance in distributed settings. Our experimental results show that our approach can scale to hundreds of workers for queries of various complexities, while processing input batches with few second latencies.

We use Spark [152, 151] to parallelize the execution of incremental view maintenance code. Spark offers a synchronous model of computation in which the driver governs job execution and coordination with workers.

We execute a subset of the TPC-H queries with different complexities on a 500GB stream of tuples. We chunk the input stream into batches of a given size, and, for each batch, we run one or more Spark jobs to refresh the materialized view. To avoid scalability bottlenecks caused by the driver handling all input data, we simulate a system in which every worker receives, independently of the driver and other workers, a fraction of the input stream. In our experiments, we ensure that each worker gets a roughly equal *random* partition of every batch. Each worker preloads its batch partitions before starting the experiment.

Materialized views are either stored locally on the driver or distributed among workers. The decision on how to partition materialized views in order to minimize their maintenance costs is a challenging problem, which might benefit from previous work on database partitioning [59,

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Jobs	1	1	1	1	2	1	3	2	3	1	2
Stages	1	3	3	2	5	1	6	6	7	3	4
	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Jobs	1	2	1	1	3	1	1	1	1	2	2
Stages	2	4	2	3	5	2	3	2	3	4	3

Table 5.1 – View maintenance complexity of TPC-H queries in Spark.

124]. We leave this question for future work. In this work, we rely on a simple heuristics rule: we partition materialized views on the primary key of a base table appearing in the view schema (e.g., `orderkey`); if there are multiple such primary keys of base tables (e.g., `orderkey`, `custkey`), we partition on the one with the highest cardinality (`orderkey`); otherwise, if there are no primary keys in the view schema, we assume the final aggregate has a small domain and can be stored on the driver.

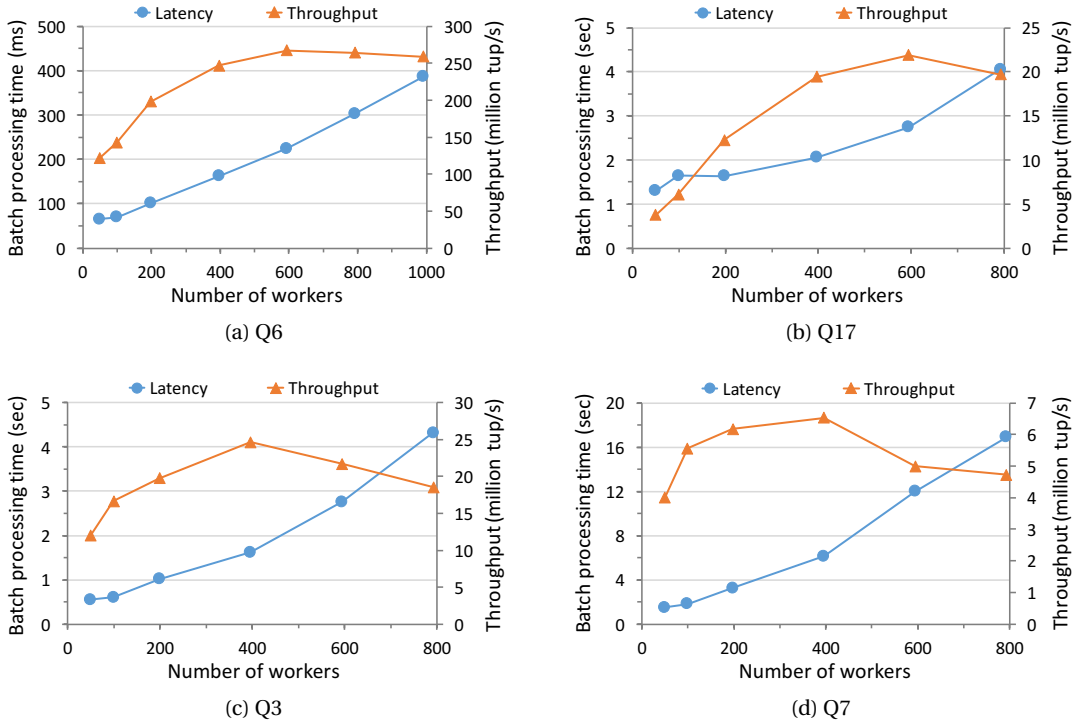
Query Complexity in Spark Generated Spark code runs a sequence of jobs to perform incremental view maintenance. Each job consists of multiple stages (e.g., map-reduce phases), and each stage corresponds to one block of distributed statements. In Table 5.1, we show the complexity of the TPC-H queries in Spark expressed as the number of jobs and stages necessary to process one batch of updates to base relations, assuming the partitioning strategy described above. The structure of each query determines the number of jobs and stages.

5.5.1 Weak Scalability

We evaluate the scalability of DBToaster when each worker receives batch partitions of size 100,000. Figure 5.6 shows the measured latency and throughput for some TPC-H queries.

Q6 computes a single aggregate over the Lineitem relation. Given the initial random distribution of batches and small query output size, we create one stage during which each worker computes a partial aggregate of its batch partition, and then, we sum these values up to update the final result at the driver. The purpose of running Q6 is to measure Spark synchronization overheads as a function of the number of workers. The query requires minimal network communication as each worker sends one 64-bit value per batch. Also each worker spends negligible time aggregating 100,000 tuples (6 ms on average), thus the results from Figure 5.6a are close to pure synchronization overheads of Spark. The median latency of processing a batch of size (100,000 × #workers) increases from 65 ms for 50 workers to 386 ms for 1,000 workers, while the throughput rises up to 267 million tuples per sec for 600 workers. Both metrics suffer from synchronization costs, which increase with more workers.

Q17 computes a two-way join with an equality-correlated nested aggregate. Incremental computation of Q17 relies on the domain extraction procedure described in Section 3.4.2.



(a) Q6

(b) Q17

(c) Q3

(d) Q7

Figure 5.6 – Weak scalability of the incremental view maintenance of TPC-H queries. Each worker processes batches of size 100,000.

We partition both base relations on `partkey` and store the result at the driver. The execution graphs consists of two stages. The first stage pre-aggregates batch partitions and shuffles the result on `partkey`. The second stage refreshes the base relations and aggregates partial results at the driver to update the final result.

Figure 5.6b shows the performance of incremental maintenance of Q17. The throughput rises up to 600 nodes, while the median latency increases from 1.3s for 50 workers to 4s for 800 workers. Q17 achieves higher latency than Q6 due to several reasons: (1) workers perform more expensive view maintenance (526 ms on average), (2) the shuffling phase requires serialization and deserialization of data, writing to local disks, and reading from remote locations, and (3) more processing stages incur more synchronization overheads. Pre-aggregation of input batches reduces the amount of shuffled data from 7.6 MB to 1.8 MB per worker. This amount remains constant regardless of the number of workers.

Figure 5.6c shows that the average throughput of Q3 increases up to 400 workers. Compared with Q17, the median latency of Q3 is lower at smaller scales and almost identical when using more workers. Q3 uses one additional stage to replicate pre-aggregated `CUSTOMER` deltas and join them with materialized views partitioned over `orderkey`. The amount of shuffled data per worker grows with the batch size, from 439 KB for 50 workers to 2.4 MB for 1,000 workers. The trigger processing time per worker (excluding all other overheads) changes on average from

120 ms for 50 workers to 305 ms for 1,000 workers, with less than 10% deviation among workers in both cases. So, at larger scales, the increased shuffling cost dominates the processing time.

Q7 is one of the most complex queries in our workload from the perspective of incremental view maintenance. The driver stores the top-level result and runs three jobs to process one batch of updates. The median latency grows more rapidly compared to other queries, from 1.5s on 50 workers to 16.9s on 800 workers. The average running time of the update triggers per worker also increases but more steadily, from 0.6s to 3.7s, while the amount of shuffled data per worker grows from 2.1 MB to 8.4 MB. This increased network communication induces higher latencies and brings down the average throughput beyond 400 workers.

In all these cases, using more workers increases the variability of latency. From our experience with using Spark, stragglers can often prolong stage computation time by a factor of 1.5 – 3x despite almost perfect load balancing. We also observe that the straggler effect is more pronounced with queries shuffling relatively large amounts of data, such as Q3 and Q7. Examining logs and reported runtime metrics gives no reasonable explanation for such behavior.

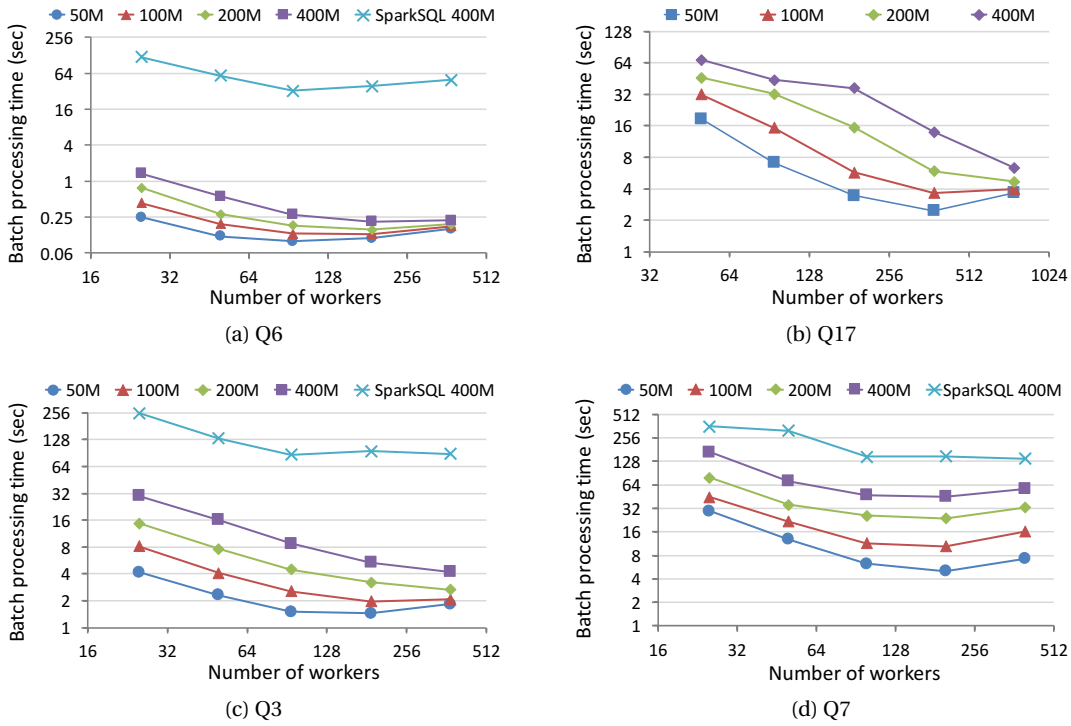
5.5.2 Strong Scalability

We measure the scalability of our incremental technique for constant batch sizes and varying numbers of workers. Figure 5.7 shows the measured throughput for a subset of the TPC-H queries. We use batches with 50, 100, 200, and 400 million tuples to ensure enough parallelizable work inside update triggers. Figure A.1 in Appendix A.2 shows results for more TPC-H queries. We compare our approach against re-evaluation using Spark SQL for batches with 400 million tuples. Note that Spark SQL can handle only flat queries.

Figure 5.7a shows that the median latency of Q6 decreases with more workers until the cost of synchronization becomes comparable with the cost of batch processing. Processing 100 million tuples using 100 workers takes on average 14 ms per worker, leaving no opportunities for further parallelization. For the four batch sizes, the lowest median latencies are 98 ms, 130 ms, 153 ms, and 211 ms. Re-evaluating Q6 on each update using Spark SQL achieves the median latency of 32.8 seconds per batch on 100 nodes.

The incremental view maintenance of Q17 scales almost linearly, as shown in Figure 5.7b. The median latency of processing one batch of 400 million tuples declines by 10.7x (from 68.5s to 6.4s) when using 16x more resources (from 50 to 800 workers). Here, the amount of shuffled data per worker decreases from 77.4 MB to 4.2 MB, while the trigger processing time per worker drops from 27.3s to 2.3s. We observe similar effects when processing smaller batches. When using 800 workers, the median latency of processing different batch sizes varies from 3.6s to 6.4s.

Figure 5.7c shows that the median latency of processing Q3 decreases with more nodes, from 30s with 25 workers to 4.2s with 400 workers for input batches with 400 million tuples. Adding more workers decreases the amount of work performed inside each trigger and the amount of



(a) Q6

(b) Q17

(c) Q3

(d) Q7

Figure 5.7 – Strong scalability of the incremental view maintenance of TPC-H queries for different batch sizes (in million of tuples). Appendix A.2 has results for more TPC-H queries.

shuffled data per worker while at the same time increases synchronization overheads. Using larger batches creates more parallelizable work and enables scalable execution across more nodes, as shown in Figure 5.7c. Re-evaluating Q3 using Spark SQL performs slower than our incremental program for the corresponding batch size, from 8.5x using 25 workers to 20.9x using 400 workers.

Q7 requires the most expensive maintenance work among the four TPC-H queries. The median latency of processing 100 million tuples drops from 44.8s with 25 workers to 10.4s with 200 workers. Beyond 200 workers, even though the size of shuffled data per worker decreases, managing large data creates stragglers that prolong execution time. Compared with Spark SQL re-evaluation, our approaches achieves 3.3x lower median latency with 200 workers.

Using fewer workers increases the variability of latency in almost all our queries due to larger amounts of shuffled data per worker. This observation confirms our previous conclusion that shuffling large data among many workers creates stragglers.

5.5.3 Optimization Effects

Figure 5.8 shows the effects of our optimizations from Section 5.3 on the distributed incremental view maintenance of TPC-H Q3 for input batches with 200 million tuples. We consider

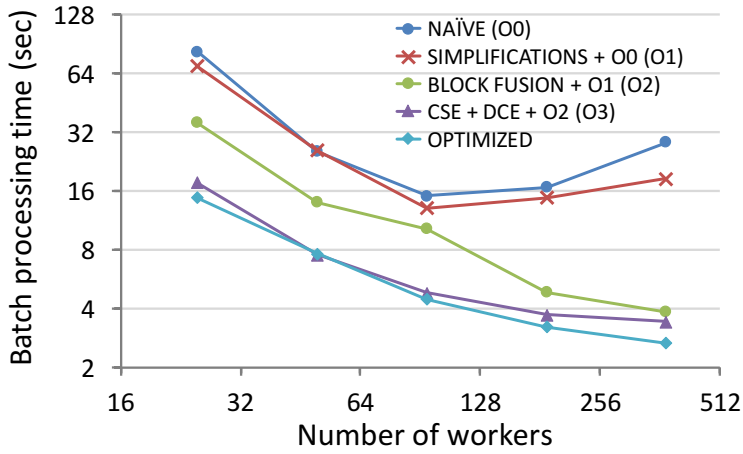


Figure 5.8 – Optimization effects on the distributed incremental view maintenance of TPC-H Q3 for batches with 200 million tuples.

the naive implementation with all optimizations turned off; then, we include simplification rules for location transforms to minimize their number, followed by enabling the block fusion algorithm. Finally, we apply CSE and DCE optimizations to eliminate trigger statements doing redundant network communication during program execution.

Our results show that applying simplification rules can reduce the median latency of incremental processing of Q3 by 35% when using 400 workers. Grouping together trigger statements using the block fusion algorithm reduces the number of stages necessary to process one input batch, which enables scalable execution. Eliminating redundant network communication statements further decreases the latency by 11% for 400 workers. The final optimized program relies on the Spark framework to pipeline processing stages, which brings up to 22% performance improvements.

5.6 Summary

We describe a novel approach for compiling incremental view maintenance code into data-parallel programs optimized for running in distributed environments. We present a set of optimizations, heuristic rules, and algorithms for building distributed view maintenance programs while trying to minimize network communication overheads. We show that our approach can scale to hundreds of workers for queries of various complexities while processing input batches with few second latencies.

6 Incremental View Maintenance of Linear Algebra Queries

Many analytics tasks and machine learning problems are naturally described using matrix algebra. In this chapter, we focus on the incremental computation of complex analytical queries written as iterative linear algebra programs. A program consists of a sequence of statements performing operations on vectors and matrices. For each matrix (vector) that dynamically changes over time, our goal is to define a trigger program describing how an incremental update affects the result of each statement (materialized view). A *delta expression* of one statement captures the difference between the new and old result. This chapter shows how to efficiently propagate delta expressions through program statements while avoiding re-evaluation of computationally expensive operations, like matrix multiplication or inversion.

6.1 Challenges and Contributions

Example 6.1.1 To demonstrate the challenges arising in incremental linear algebra, let us consider a program that computes the fourth power of a given matrix A. The program consists of two consecutive statements:

Listing 6.1 Linear algebra program for computing A^4

```
1  B := A A;
2  C := B B;
```

Our goal is to maintain the result C on every update of A by ΔA. We compare the time complexity of two computation strategies: re-evaluation and incremental maintenance. The re-evaluation strategy first applies ΔA to A and then performs two $\mathcal{O}(n^3)$[1] matrix multiplications to update C.

The incremental approach exploits the associativity and distributivity of matrix multiplication to compute a delta expression for each statement of the program. The trigger program for

[1] This example assumes the traditional cubic-time bound for matrix multiplication. Section 6.2 generalizes this cost for asymptotically more efficient methods.

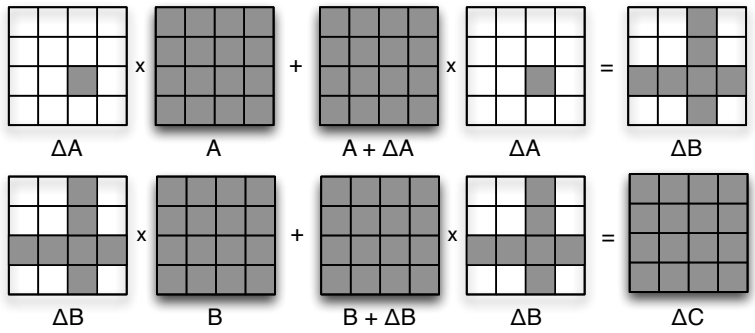


Figure 6.1 – A graphical representation of the evaluation of ΔB and ΔC for a single entry change in A. Gray entries have nonzero values.

updates to A is:

Listing 6.2 Incremental linear algebra program for maintaining A^4

```
1  ON UPDATE A BY ΔA:
2    ΔB := (ΔA)A + A(ΔA) + (Δ)A)(ΔA);
3    ΔC := (ΔB)B + B(ΔB) + (Δ)B)(ΔB);
4    A += ΔA; B += ΔB; C += ΔC;
```

Let us assume that ΔA represent a change of one cell in A. Figure 6.1 shows the effect of that change on ΔB and ΔC[2]. The shaded regions represent entries with nonzero values. The incremental approach capitalizes on the sparsity of ΔA and ΔB to compute ΔB and ΔC in $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ operations, respectively. Together with the cost of updating B and C, incremental evaluation of the program requires $\mathcal{O}(n^2)$ operations, clearly cheaper than re-execution. □

The main challenge in incremental linear algebra is how to represent and propagate delta expressions. Even a small change in a matrix (e.g., a change of one entry) might have an avalanche effect that updates every entry of the result matrix. In Example 6.1.1, a single entry change in A causes changes of one row and column in B, which in turn pollute the entire matrix C. If we were to propagate ΔC to a subsequent expression – for example, the expression $D = CC$ that computes A^8 – then evaluating ΔD would require two full $\mathcal{O}(n^3)$ matrix multiplications, which is obviously more expensive than recomputing D using the new value of C.

To confine the effect of such changes and allow efficient evaluation, we represent delta expressions in a *factored form*, as products of low-rank matrices. For instance, we could represent ΔB as a vector outer product for single entry updates in A. Due to the associativity and distributivity of matrix multiplication, the factored form allows us to choose the evaluation order that completely avoids expensive matrix multiplications.

[2] For brevity, we factor the last two monomials in ΔB and ΔC.

To the best of our knowledge, this is the first work done towards efficient incremental computation for large scale linear algebra data analysis programs. In brief, our contribution can be summarized as follows:

1. We present a framework for incremental maintenance of linear algebra programs that: (a) represents delta expressions in compact factored forms that confine the avalanche effect of input changes, as seen in Example 6.1.1, and thus cost; (b) utilizes a set of transformation rules that metamorphose linear algebra programs into their cheap functional equivalents that are optimized for dynamic datasets.

2. We demonstrate analytically and experimentally the efficiency of incremental processing on various fundamental data analysis methods including ordinary least squares, batch gradient descent, PageRank, and matrix powers.

3. We have built LINVIEW, a compiler for incremental data analysis that exploits these novel techniques to generate efficient update triggers optimized for dynamic datasets. The compiler is easily extensible to couple with any underlying system that supports matrix manipulation primitives. We evaluate the performance of LINVIEW's generated code over two different platforms: (a) Octave programs running on a single machine, and (b) parallel Spark programs running over a large cluster of Amazon EC2 nodes. Our results show that incremental evaluation provides an order of magnitude performance benefit over traditional re-evaluation.

This chapter is organized as follows. Section 6.2 establishes the terminology and computational models used in this chapter, Section 6.3 describes how to compute, represent, and propagate delta expressions, Section 6.4 analyzes the efficiency of incremental maintenance on common data analytics, Section 6.5 gives the system overview, Section 6.6 experimentally validates our analysis, and Section 6.7 concludes this chapter.

6.2 Linear Algebra Programs

Linear algebra programs express computations using vectors and matrices as high-level abstractions. The language used to form such programs consists of the standard matrix manipulation primitives: matrix addition, subtraction, multiplication (including scalar, matrix-vector, and matrix-matrix multiplication), transpose, and inverse. A program expresses a computation as a sequence of statements, each consisting of an expression and a variable (matrix) storing its result. The program evaluates these expressions on a given dataset of *input matrices* and produces the result in one or more *output matrices*. The remaining matrices are auxiliary program matrices, which can be manipulated (materialized or removed) for performance reasons. For instance, the program of Example 6.1.1 consists of two statements evaluating the expressions over an input matrix A and an auxiliary matrix B. The output matrix C stores the computation result.

6.2.1 Computational complexity

In this thesis, we refer to the cost of square matrix multiplication as $\mathcal{O}(n^\gamma)$, where $2 \leq \gamma \leq 3$. In practice, the complexity of matrix multiplication, for example, BLAS implementations [148], is bounded by cubic $\mathcal{O}(n^3)$ time. Better algorithms with an exponent of $2.37 + \epsilon$ are known (Coppersmith-Winograd and its successors); however, these algorithms are only relevant for astronomically large matrices. Our incremental techniques remain relevant as long as matrix multiplication stays asymptotically worse than quadratic time (a bound that has been conjectured to be achievable [53], but still seems far off). Note that the asymptotic lower bound for matrix multiplication is $\Omega(n^2)$ operations because it needs to process at least $2n^2$ entries.

6.2.2 Iterative Programs

Many computational problems are iterative in nature. Iterative programs start from an approximate answer, and each iteration step improves the accuracy of the solution until the estimated error drops below a specified threshold. Iterative methods are often the only choice for problems for which direct solutions are either unknown (e.g., nonlinear equations) or prohibitively expensive to compute (e.g., due to large problem dimensions).

In this work, we study (iterative) linear algebra programs from the viewpoint of incremental view maintenance (IVM). The execution of an iterative program generates a sequence of results, one for each iteration step. When the underlying data changes, IVM updates these results rather than re-evaluating them from scratch. We do so by propagating the delta expression of one iteration to subsequent iterations. With our incremental techniques, such delta expressions are cheaper to evaluate than the original expressions.

We consider iterative programs that execute a fixed number of iteration steps. The reason for this decision is that programs using convergence thresholds might yield a varying number of iteration steps after each update. Having different numbers of outcomes per update would require incremental maintenance to deal with outdated or missing old results; we leave this topic for future work. By fixing the number of iterations, we provide a fair comparison of the incremental and re-evaluation strategies [3].

6.2.3 Iterative Models

An iterative computation is governed by an iterative function that describes the computation at each step in terms of the results of previous iterations (materialized views) and a set of input matrices. Multiple iterative functions, or *iterative models*, might express the same computation but by using different numbers of iteration steps. For instance, the computation of the k^{th} power of a matrix can be done in k iterations or in $\log_2 k$ iterations using the exponentiation by squaring method. Each iterative model of computation comes with its own complexity.

[3] If the solution does not converge after a given number of iterations, we can always re-evaluate additional steps.

Our analysis of iterative programs considers three alternative models that require different numbers of iteration steps to compute the final result. These models allow us to explore trade-offs between computation time and memory consumption for both re-evaluation and incremental maintenance.

Linear Model

The linear iterative model evaluates the result of the current iteration based on the result of the previous iteration and a set of input matrices $\mathcal{I}$. It takes k iteration steps to compute T_k.

$$T_i = \begin{cases} f(\mathcal{I}) & \text{for } i = 1 \\ g(T_{i-1}, \mathcal{I}) & \text{for } i = 2,3,\ldots \end{cases}$$

For example, the linear iterative model for computing the k-th power of a given matrix A is $T_1 = A$ and $T_i = T_{i-1}\,A$, for $2 \leq i \leq k$.

Exponential Model

In the exponential model, the result of the i^{th} iteration depends on the result of the $(i/2)^{th}$ iteration. The model makes progressively larger steps between computed iterations, forming the sequence T_1, T_2, T_4, ... It takes $\mathcal{O}(\log k)$ iteration steps to compute T_k.

$$T_i = \begin{cases} f(\mathcal{I}) & \text{for } i = 1 \\ g(T_{i/2}, \mathcal{I}) & \text{for } i = 2,4,8\ldots \end{cases}$$

For example, the exponential iterative model for computing A^k when k is a power of two is $T_1 = A$ and $T_i = T_{i/2}\,T_{i/2}$ for $i = 2,4,8,\ldots,k$.

Skip Model

Depending on the dimensions of input matrices, incremental evaluation using the above models might be suboptimal costwise. The skip-s model represents a sweet spot between these two models. For a given skip size s, it relies on the exponential model to compute T_s (generating the sequence T_2, T_4, ..., T_s) and then generalizes the linear model to compute every s^{th} iteration (generating the sequence T_{2s}, T_{3s}, ...).

$$T_i = \begin{cases} f(\mathcal{I}) & \text{for } i = 1 \\ g(T_{i/2}, \mathcal{I}) & \text{for } i = 2,4,8,\ldots s \\ h(T_{i-s}, T_s, \mathcal{I}) & \text{for } i = 2s,3s,\ldots \end{cases}$$

For example, the skip iterative model for computing A^k when $s = 8$ and k is divisible by s is $T_1 = A$, then $T_i = T_{i/2}\,T_{i/2}$ for $i = 2,4,8$, and $T_i = T_{i-8}\,T_8$ for $i = 16,24,32,\ldots,k$.

The skip model reconciles the two extremes: it corresponds to the linear model for $s = 1$ and to the exponential model for $s = k$. In Section 6.4, we evaluate the time and space complexity of these models for a set of iterative programs.

6.3 Incremental Processing

In this section, we develop techniques for converting linear algebra programs into functionally equivalent *incremental programs* suited for execution on dynamic datasets. An incremental program consists of a set of triggers, one trigger for each input matrix that might change over time. Each trigger has a list of update statements that maintain the result for updates to the associated input matrix. The total execution cost of an incremental program is the sum of execution costs of its triggers. Incremental programs incur lower computational complexity by converting the expensive operations of non-incremental programs to work with smaller datasets. Incremental programs combine precomputed results with low-rank updates to avoid costly operations, like matrix-matrix multiplications or matrix inversions.

Definition 6.3.1 *A matrix $\mathcal{M}$ of dimensions $(n \times n)$ is said to have rank-k if the maximum number of linearly independent rows or columns in the matrix is k. $\mathcal{M}$ is called a low-rank matrix if $k \ll n$.*

6.3.1 Delta Derivation

The basic step in building incremental programs is the derivation of *delta expressions* $\Delta_A(E)$, which capture how the result of an expression E changes as an input matrix A is updated by ΔA. We consider the update ΔA, called a *delta matrix*, to be constant and independent of any other matrix. If we represent E as a function of A, then $\Delta_A(E) = E(A + \Delta A) - E(A)$. For presentation clarity, we omit the subscript in $\Delta_A(E)$ when A is obvious from the context.

Most standard operations of linear algebra are amenable to incremental processing. Using the distributive and associative properties of common matrix operations, we derive the following set of delta rules for updates to A:

$$
\begin{aligned}
\Delta_{\mathrm{A}}(E_1\,E_1) &:= (\Delta_{\mathrm{A}}E_1)\,E_1 + E_1\,(\Delta_{\mathrm{A}}E_1) + (\Delta_{\mathrm{A}}E_1)\,(\Delta_{\mathrm{A}}E_1)\\
\Delta_{\mathrm{A}}(E_1 \pm E_1) &:= (\Delta_{\mathrm{A}}E_1) \pm (\Delta_{\mathrm{A}}E_1)\\
\Delta_{\mathrm{A}}(\lambda\,E) &:= \lambda\,(\Delta_{\mathrm{A}}E)\\
\Delta_{\mathrm{A}}(E^{\mathrm{T}}) &:= (\Delta_{\mathrm{A}}E)^{\mathrm{T}}\\
\Delta_{\mathrm{A}}(E^{-1}) &:= (E + \Delta_{\mathrm{A}}E)^{-1} - E^{-1}\\
\Delta_{\mathrm{A}}(A) &:= \Delta A\\
\Delta_{\mathrm{A}}(B) &:= 0 \qquad (A \neq B)
\end{aligned}
$$

where λ is a scalar. Note that $\Delta_{\mathrm{A}}(\Delta A) = 0$.

We observe that the delta rule for matrix inversion references the original expression (twice), which implies that it is more expensive to compute the delta expression than the original expression. This claim is true for arbitrary updates to A (e.g., random updates of all entries, all done at once). Later on, we discuss a special form of updates that admits efficient incremental maintenance of matrix inversions. Note that if A does not appear in E, the delta expression for matrix inversion is zero.

Example 6.3.2 This example shows the derivation process. Consider the Ordinary Least Squares method for estimating the unknown parameters in a linear regression model. We want to find a statistical estimate of the parameter β^* best satisfying $Y = X\beta$. The solution, written as a linear algebra program, is $\beta^* = (X^\mathrm{T} X)^{-1} X^\mathrm{T} Y$. Here, we focus on how to derive the delta expression for β^* under updates to X. We defer an in-depth cost analysis of the method to Section 6.4. Let $Z = X^\mathrm{T} X$ and $W = Z^{-1}$, then

$$\begin{aligned}\Delta Z &= \Delta(X^\mathrm{T} X)\\ &= (\Delta(X^\mathrm{T}))\, X + X^\mathrm{T}\,(\Delta(X)) + (\Delta(X^\mathrm{T}))\,(\Delta(X))\\ &= (\Delta X)^\mathrm{T} X + X^\mathrm{T}\,(\Delta X) + (\Delta X)^\mathrm{T}\,(\Delta X)\end{aligned}$$

and $\Delta W = (Z + \Delta Z)^{-1} - Z^{-1}$. Finally, $\Delta\beta^* = (\Delta W)\, X^\mathrm{T} Y + W\,(\Delta X)^\mathrm{T} Y + (\Delta W)\,(\Delta X)^\mathrm{T} Y$. □

For random matrix updates, incrementally computing a matrix inverse is prohibitively expensive. The Sherman-Morrison formula [127] provides a numerically cheap way of maintaining the inverse of an invertible matrix for rank-1 updates. Given a rank-1 update $u\,v^\mathrm{T}$, where u and v are column vectors, if E and $E + u\,v^\mathrm{T}$ are nonsingular, then

$$\Delta(E^{-1}) = -\frac{E^{-1}\,u\,v^\mathrm{T}\,E^{-1}}{1 + v^\mathrm{T} E^{-1} u}$$

Note that $\Delta(E^{-1})$ is also a rank-1 matrix. For instance, $\Delta(E^{-1}) = p\,q^\mathrm{T}$, where $p = \lambda\,E^{-1}u$ and $q = (E^{-1})^\mathrm{T} v$ are column vectors, and λ is a scalar (observe that the denominator is a scalar too). Thus, incrementally computing E^{-1} for rank-1 updates to E requires $\mathcal{O}(n^2)$ operations; it avoids any matrix-matrix multiplication and inversion operations.

Example 6.3.3 We apply the Sherman-Morrison formula to Example 6.3.2. We start by considering rank-1 updates to X. Let $\Delta X = u\,v^\mathrm{T}$, then

$$\begin{aligned}\Delta Z &= v\,u^\mathrm{T} X + X^\mathrm{T} u\,v^\mathrm{T} + v\,u^\mathrm{T} u\,v^\mathrm{T}\\ &= v\,(u^\mathrm{T} X) + (X^\mathrm{T} u + v\,u^\mathrm{T} u)\,v^\mathrm{T}\end{aligned}$$

The parentheses denote the subexpressions that evaluate to vectors. We observe that each of the monomials is a vector outer product. Thus, we can write $\Delta Z = \Delta Z_1 + \Delta Z_2$, where $\Delta Z_1 = p_1\,q_1^\mathrm{T}$ and $\Delta Z_2 = p_2\,q_2^\mathrm{T}$. Now we can apply the formula on each outer product in turn.

$$\Delta_{Z_1}(W) = -\frac{W\,p_1\,q_1^{\mathrm{T}}\,W}{1+q_1^{\mathrm{T}}\,W\,p_1} \qquad \Delta_{Z_2}(W) = -\frac{(W+\Delta_{Z_1}(W))\,p_2\,q_2^{\mathrm{T}}\,(W+\Delta_{Z_1}(W))}{1+q_2^{\mathrm{T}}\,(W+\Delta_{Z_1}(W))\,p_2}$$

Finally, $\Delta_Z(W) = \Delta_{Z_1}(W) + \Delta_{Z_2}(W)$. Computing $\Delta_Z(W)$ requires $\mathcal{O}(n^2)$ operations, as discussed above. For comparison, the evaluation cost of ΔW in Example 6.3.2 is $\mathcal{O}(n^\gamma)$. □

In the above OLS examples, ΔW is a matrix with potentially all nonzero entries. If we store these entries in a single delta matrix, we still need to perform $\mathcal{O}(n^\gamma)$-cost matrix multiplications in order to compute $\Delta\beta^*$. Next, we propose an alternative way of representing delta expressions that allows us to stay in the realm of $\mathcal{O}(n^2)$ computations.

6.3.2 Delta Representation

In this section, we discuss how to represent delta expressions in a form that is amenable to incremental processing. This form also dictates the structure of admissible updates to input matrices. Incremental processing brings no benefit if the whole input matrix changes arbitrarily at once.

Let us consider updates of the smallest granularity – single entry changes of an input matrix. The *delta matrix* capturing such an update contains exactly one nonzero entry being updated. The following example shows that even a minor change, when propagated naïvely, can cause incremental processing to be more expensive than recomputation.

Example 6.3.4 Consider the program of Example 6.1.1 for computing the fourth power of a given matrix A. Following the delta rules we write $\Delta B = (\Delta A)\,A + (A+\Delta A)\,(\Delta A)$. Figure 6.1 shows the effect of a single-entry change in A on ΔB. The change has escalated to a change of one row and one column in B. When we propagate this change to the next statement, ΔC becomes a fully-perturbed delta matrix where all entries might have nonzero values.

Now, suppose we want to evaluate A^8, so we extend the program with the statement $D := C\,C$. To evaluate ΔD, which is expressed similarly as ΔB, we need to perform two matrix-matrix multiplications and two matrix additions. Clearly, in this case, it is more efficient to recompute D using the new C than to incrementally maintain it with ΔD. □

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 its extreme after executing merely two statements. This observation suggests $\mathcal{O}(n^2)$ is the lower bound of this computation.

We propose a novel approach to deal with escalating updates. So far, we have used a single matrix to store the result of a delta expression. We observe that such representation is highly redundant as delta matrices typically have low ranks. Although a delta matrix might contain

all nonzero entries, the number of linearly independent rows or columns is relatively small compared to the matrix size. In Example 6.3.4, ΔB has a rank of at most two.

We maintain a delta matrix in a *factored form*, represented as a product of two low-rank matrices. The factored form enables more efficient evaluation of subsequent delta expressions. Due to the associativity and distributivity of matrix multiplication, we can base the evaluation strategy for delta expressions solely on matrix-vector products, and thus avoid expensive matrix-matrix multiplications.

To achieve this goal, we also represent updates of input matrices in the factored form. In this thesis, we consider rank-k changes of input matrices as they can capture many practical update patterns. For instance, the simplest rank-1 updates can express perturbations of one complete row or column in a matrix, or even changes of the whole matrix when the same vector is added to every row or column.

In Example 6.3.4, consider a rank-1 update $\Delta A = u_A v_A^T$, where u_A and v_A are column vectors, then $\Delta B = u_A (v_A^T A) + (A u_A) v_A^T + (u_A v_A^T u_A) v_A^T$ is a sum of three outer products. The parentheses denote the factored subexpressions (vectors). The evaluation order enforced by these parentheses yields only matrix-vector and vector-vector multiplications. Thus, the evaluation of ΔB requires only $\mathcal{O}(n^2)$ operations.

Instead of representing delta expressions as sums of outer products, we maintain them in a more compact vectorized form for performance and presentation reasons. A sum of k outer products is equivalent to a single product of two matrices of sizes $(n \times k)$ and $(k \times n)$, which are obtained by stacking the corresponding vectors together. For instance,

$$u_1 v_1^T + u_2 v_2^T + u_3 v_3^T = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ v_3^T \end{bmatrix} = P Q^T$$

where P and Q are $(n \times 3)$ block matrices.

To summarize, we maintain a delta expression as a product of two low-rank matrices with dimensions $(n \times k)$ and $(k \times n)$, where $k \ll n$. This representation allows efficient evaluation of subsequent delta expressions without involving expensive $\mathcal{O}(n^\gamma)$ operations; instead, we perform only $\mathcal{O}(kn^2)$ operations. A similar analysis naturally follows for rank-k updates with linearly increasing evaluation costs; the benefit of incremental processing diminishes as k approaches the dominant matrix dimension.

Considering low-rank updates of matrices also opens opportunities to benefit from previous work on incrementalizing complex linear algebra operations. We have already discussed the Sherman-Morrison method of incrementally computing the inverse of a matrix for rank-1 updates. Other work [63, 134] investigates rank-1 updates in different matrix factorizations, like SVD and Cholesky decomposition. We can further use these new primitives to enrich our language, and, consequently, support more sophisticated programs.

6.3.3 Delta Propagation

When constructing incremental programs we propagate delta expressions from one statement to another. For delta expressions with multiple monomials, factored representations include increasingly more outer products. That raises the cost of evaluating these expressions. In Example 6.3.4, ΔB consists of three outer products compacted as

$$\Delta B = \begin{bmatrix} u_A & (A\,u_A) & (u_A\,(v_A^T\,u_A)) \end{bmatrix} \begin{bmatrix} v_A^T A \\ v_A^T \\ v_A^T \end{bmatrix} = U_B\,V_B^T$$

Here, U_B and V_B are $(n \times 3)$ block matrices. Akin to ΔB, ΔC is also a sum of three products expressed using B, U_B, and V_B, and compacted as a product of two $(n \times 9)$ block matrices. Finally, we use C and the factored form of ΔC to express ΔD as a product of two $(n \times 27)$ block matrices.

Observe that U_B and V_B have linearly dependent columns, which suggests that we could have an even more compact representation of these matrices. A less redundant form, which reduces the size of U_B and V_B, guarantees less work in evaluating subsequent delta expressions. To alleviate the redundancy in representation, we reduce the number of monomials in a delta expression by extracting common factors among them. This syntactic approach does not guarantee the most compact representation of a delta expression, which is determined by the rank of the delta matrix. However, computing the exact rank of the delta matrix requires inspection of the matrix values, which we deem too expensive. The factored form of ΔB of Example 6.3.4 is

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

Here, U_B and V_B are $(n \times 2)$ matrices. ΔC is a product of two $(n \times 4)$ matrices and ΔD multiplies two $(n \times 8)$ matrices.

6.3.4 Putting It All Together

So far, we have discussed how to derive, represent, and propagate delta expressions. In this section, we put these techniques together into an algorithm that compiles a given program to its incremental version.

Algorithm 6.3 shows the procedure that transforms a program $\mathcal{P}$ into a set of trigger functions $\mathcal{T}$, each of them handling updates to one input matrix. For updates arriving as vector outer products, the matching trigger incrementally maintains the computation result by evaluating a sequence of assignment statements (:=) and update statements (+=).

The algorithm takes as input two parameters: (1) a program $\mathcal{P}$ expressed as a list of assignment statements, where each statement is defined as a tuple $\langle A_i, E_i \rangle$ of an expression E_i and a matrix A_i storing the result, and (2) a set of input matrices $\mathcal{I}$; it outputs a set of trigger functions $\mathcal{T}$.

Algorithm 6.3 Compile program $\mathcal{P}$ into a set of triggers $\mathcal{T}$

```
1  Input: Program P consisting of statements A_i := E_i and a set of input matrices I
2  Output: Triggers T maintaining the output on changes of each input matrix
3  function COMPILE(P, I)
4      T ← ∅
5      for each X ∈ I do
6          D ← list(⟨X, u, v⟩)
7          for each ⟨A_i, E_i⟩ ∈ P do
8              ⟨P_i, Q_i⟩ ← COMPUTEDELTA(E_i, D)
9              D ← D.append(⟨A_i, P_i, Q_i⟩)
10         end for
11         T ← T ∪ BUILDTRIGGER(X, D)
12     end for
13     return T
14 end function
```

The COMPUTEDELTA function follows the rules from Section 6.3.1 to derive the delta for a given expression E_i and an update to X. The function returns two expressions that together form the delta, $\Delta A_i = P_i Q_i^{\mathrm{T}}$. As discussed in Section 6.3.2, P_i and Q_i are block matrices in which each block has its defining expression.

The algorithm maintains a list of the generated delta expressions in $\mathcal{D}$. Each entry in $\mathcal{D}$ corresponds to one update statement of the trigger program. The entries respect the order of statements in the original program.

Note that COMPUTEDELTA takes $\mathcal{D}$ as input. The list of matrices affected by a change in X – initially containing only X – expands throughout the execution of the algorithm. One expression might reference more than one such matrix, so we have to deal with multiple matrix updates to derive the correct delta expression. The delta rules presented in Section 6.3.1 consider only single matrix updates, but we can easily extend them to handle multiple matrix updates. Suppose $\mathcal{D} = \{A, B, \ldots\}$ is a set of the affected matrices that also appear in an expression E. Then, $\Delta_{\mathcal{D}}(E) := \Delta_A(E) + \Delta_{(\mathcal{D}\setminus\{A\})}(E + \Delta_A(E))$. The delta rule considers each matrix update in turn. The order of applying the matrix updates is irrelevant.

Example 6.3.5 Consider the expression $E = AB$ and the updates ΔA and ΔB. Then,

$$\begin{aligned}\Delta_{\{A,B\}}(E) &= \Delta_A(E) + \Delta_B(E + \Delta_A(E)) \\ &= (\Delta A)\,B + \Delta_B(AB + (\Delta A)\,B) \\ &= (\Delta A)\,B + A\,(\Delta B) + (\Delta A)\,(\Delta B)\end{aligned}$$

□

The BUILDTRIGGER function converts the derived deltas $\mathcal{D}$ for updates to X into a trigger program. The function first generates the assignment statements that evaluate P_i and Q_i for each delta expression, and then the update statements for each of the affected matrices.

Example 6.3.6 Consider the program that computes the fourth power of a given matrix A, discussed in Example 6.1.1 and Example 6.3.4. Algorithm 6.3 compiles the program and produces the following trigger for updates to A.

Listing 6.4 Incremental linear algebra program for computing A^4 using factored deltas

```
1 ON UPDATE A BY (u_A,v_A):
2   U_B := [ u_A    (Au_A + u_A(v_A^T u_A)) ];
3   V_B := [ (A^T v_A)   v_A ];
4   U_C := [ U_B    (BU_B + U_B(V_B^T U_B)) ];
5   V_C := [ (B^T V_B)  V_B ];
6   A += u_A v_A^T;   B += U_B V_B^T;   C += U_C V_C^T;
```

Here, u_A and v_A are column vectors, U_B, V_B, U_C, and V_C are block matrices. Each delta, including the input change, is a product of two low-rank matrices. □

6.4 Incremental Analytics

In this section, we analyze a set of programs that have wide application across many domains from the perspective of incremental maintenance. We study the time and space complexity of both re-evaluation and incremental evaluation over dynamic datasets. We show analytically that, in most of these examples, incremental maintenance exhibits better asymptotic behavior than re-evaluation in terms of execution time. In other cases, a combination of the two strategies offers the lowest time complexity. Note that our incremental techniques are general and apply to a broader range of linear algebra programs than those presented here.

6.4.1 Ordinary Least Squares

Ordinary Least Squares (OLS) is a classical method for fitting a curve to data. The method finds a statistical estimate of the parameter β^* best satisfying $Y = X\beta$. Here, $X = (m \times n)$ is a set of predictors, and $Y = (m \times p)$ is a set of responses that we wish to model via a function of X with parameters β. The best statistical estimate is $\beta^* = (X^T X)^{-1} X^T Y$. Data practitioners often build regression models from incomplete or inaccurate data to gain preliminary insights about the data or to test their hypotheses. As new data points arrive or measurements become more accurate, incremental maintenance avoids expensive reconstruction of the whole model, saving time and frustration.

First, consider the cost of incrementally computing the matrix inverse for changes in X. Let $Z = X^T X$, $W = Z^{-1}$, and $\Delta X = u\, v^T$. As derived in Example 6.3.2,

$$\Delta Z = \begin{bmatrix} v & (X^T u + v\, u^T u) \end{bmatrix} \begin{bmatrix} u^T X \\ v^T \end{bmatrix} = \begin{bmatrix} p_1 & p_2 \end{bmatrix} \begin{bmatrix} q_1^T \\ q_2^T \end{bmatrix}$$

The cost of computing p_2 and q_1 is $\mathcal{O}(mn)$. The vectors p_1, q_1, p_2, and q_2 have size $(n \times 1)$.

As shown in Example 6.3.3, we could represent the delta expressions of W as a sum of two outer products, $\Delta_{Z_1}(W) = r_1\, s_1^{\mathrm{T}}$ and $\Delta_{Z_2}(W) = r_2\, s_2^{\mathrm{T}}$; for instance, $s_1 = W^{\mathrm{T}}\, q_1$ and r_1 is the remaining subexpression in $\Delta_{Z_1}(W)$. The computation of r_1, q_1, r_2, and q_2 involves only matrix-vector $\mathcal{O}(n^2)$ operations. Then, the overall cost of incremental maintenance of W is $\mathcal{O}(n^2 + mn)$. For comparison, re-evaluation of W takes $\mathcal{O}(n^\gamma + mn^2)$ operations.

Finally, we compute $\Delta\beta^*$ for updates $\Delta X = u\, v^{\mathrm{T}}$ and $\Delta W = R\, S^{\mathrm{T}}$, where $R = \begin{bmatrix} r_1 & r_2 \end{bmatrix}$ and $S = \begin{bmatrix} s_1 & s_2 \end{bmatrix}$ are $(n \times 2)$ block matrices and $\Delta\beta^* = R\, S^{\mathrm{T}}\, X^{\mathrm{T}}\, Y + W\, v\, u^{\mathrm{T}}\, Y + R\, S^{\mathrm{T}}\, v\, u^{\mathrm{T}}\, Y$. The optimum evaluation order for this expression depends on the size of X and Y. In general, the cost of incremental maintenance of β^* is $\mathcal{O}(n^2 + mp + np + mn)$. For comparison, re-evaluation of β^* takes $\mathcal{O}(mnp + n^2 \min(m, p))$ operations.

Overall, considering both phases, the incremental maintenance of β^* for updates to X has lower computation complexity than re-evaluation. This holds even when Y is of small dimension (e.g., vector); the matrix inversion cost still dominates in the re-evaluation method. The space complexity of both strategies is $\mathcal{O}(n^2)$.

6.4.2 Matrix Powers

Our next analysis includes the computation of A^k of a square matrix A for some fixed $k > 0$. Matrix powers play an important role in many different domains including computing the stochastic matrix of a Markov chain after k steps, solving systems of linear differential equations using matrix exponentials, answering graph reachability queries where k represents the maximum path length, and computing PageRank using the power method.

Matrix powers also provide the basis for the incremental analysis of programs having more general forms of iterative computation. In such programs we often decide to evaluate several iteration steps at once for performance reasons, and matrix powers allow us to express these compound transformations between iterations, as shown later on.

Iterative Models

Table 6.1 expresses the matrix power computation using the iterative models presented in Section 6.2. In all cases, A is an input matrix that changes over time, and P_k contains the final result A^k. The linear model computes the result of every iteration, while the exponential model makes progressively larger leaps between consecutive iterations evaluating only $\log_2 k$ results. The skip model precomputes A^s in P_s using the exponential model and then reuses P_s to compute every s^{th} subsequent iteration.

Expressing the matrix power computation as an iterative process eases the complexity analysis of both re-evaluation and incremental maintenance, which we show next.

Model	Matrix Powers	Sums of Matrix Powers	General form: $T_{i+1} = A\,T_i + B$
Linear	$P_i = \begin{cases} A & \\ A\,P_{i-1} & \end{cases}$	$S_i = \begin{cases} I & \\ A\,S_{i-1} + I & \end{cases}$	$T_i = \begin{cases} A\,T_0 + B & \text{for } i = 1 \\ A\,T_{i-1} + B & \text{for } i = 2,3,\ldots,k \end{cases}$
Exponential	$P_i = \begin{cases} A & \\ P_{i/2}\,P_{i/2} & \end{cases}$	$S_i = \begin{cases} I & \\ P_{i/2}\,S_{i/2} + S_{i/2} & \end{cases}$	$T_i = \begin{cases} A\,T_0 + B & \text{for } i = 1 \\ P_{i/2}\,T_{i/2} + S_{i/2}\,B & \text{for } i = 2,4,8,\ldots,k \end{cases}$
Skip-s	$P_i = \begin{cases} A & \\ P_{i/2}\,P_{i/2} & \\ P_s\,P_{i-s} & \end{cases}$	$S_i = \begin{cases} I & \\ P_{i/2}\,S_{i/2} + S_{i/2} & \\ P_s\,S_{i-s} + S_s & \end{cases}$	$T_i = \begin{cases} A\,T_0 + B & \text{for } i = 1 \\ P_{i/2}\,T_{i/2} + S_{i/2}\,B & \text{for } i = 2,4,8,\ldots,s \\ P_s\,T_{i-s} + S_s\,B & \text{for } i = 2s,3s,\ldots,k \end{cases}$

Table 6.1 – The computation of matrix powers, sums of matrix powers, and the general iterative computation expressed as recurrence relations. For simplicity of the presentation, we assume that $\log_2 k$, $\log_2 s$, and $\frac{k}{s}$ are integers.

Cost Analysis

We analyze the time and space complexity of re-evaluation and incremental maintenance of P_k for rank-1 updates to A, denoted by $\Delta A = u\,v^{\mathrm{T}}$. We assume that A is a dense square matrix of size $(n \times n)$.

Re-evaluation Table 6.2 shows the time complexity of re-evaluating P_k in different iterative models. The re-evaluation strategy first updates A by ΔA and then recomputes P_k using the new value of A. All three models perform one $\mathcal{O}(n^\gamma)$ matrix-matrix multiplication per iteration. The total execution cost thus depends on the number of iteration steps: The exponential method clearly requires the fewest iterations $\log_2 k$, followed by $(\log_2 s + \frac{k}{s})$ and k iterations of the skip and linear models.

Table 6.2 also shows the space complexity of re-evaluation in the three iterative models. The memory consumption of these models is independent of the number of iterations. At each iteration step these models use at most two previously computed values, but not the full history of P_i values.

Incremental Maintenance This strategy captures the change in the result of every iteration as a product of two low-rank matrices $\Delta P_i = U_i\,V_i^{\mathrm{T}}$. The size of U_i and V_i and, in general, the rank of ΔP_i grow linearly with every iteration step. We consider the case when $k \ll n$ in which we can profit from the low-rank delta representation. This is a realistic assumption as many practical computations consider large matrices and relatively few iterations; for example, 80.7% of the pages in a PageRank computation converge in less than 15 iterations [87].

Table 6.2 shows the time complexity of incremental maintenance of P_k in the three iterative models. Incremental maintenance exhibits better asymptotic behavior than re-evaluation in all three models, with the exponential model clearly dominating the others.

	Model	(Sums of) Matrix Powers		General form: $T_{i+1} = A\,T_i + B$		
		Re-eval	Incremental	Re-eval	Incremental	Hybrid
Time	Linear	$n^\gamma k$	$n^2 k^2$	$pn^2 k$	$(n^2 + pn)k^2$	$pn^2 k$
	Exp	$n^\gamma \log k$	$n^2 k$	$(n^\gamma + pn^2)\log k$	$(n^2 + pn)k$	$pn^2 \log k + n^2 k$
	Skip-s	$n^\gamma(\log s + \frac{k}{s})$	$n^2 \frac{k^2}{s}$	$n^\gamma \log s + pn^2(\log s + \frac{k}{s})$	$(n^2 + np)\frac{k^2}{s}$	$pn^2(\log s + \frac{k}{s}) + n^2 s$
Space	Linear	n^2	$n^2 k$	$n^2 + np$	$n^2 + knp$	$n^2 + knp$
	Exp	n^2	$n^2 \log k$	$n^2 + np$	$(n^2 + np)\log k$	$(n^2 + np)\log k$
	Skip-s	n^2	$n^2(\log s + \frac{k}{s})$	$n^2 + np$	$(n^2 + np)\log s + np\frac{k}{s}$	$(n^2 + np)\log s + np\frac{k}{s}$

Table 6.2 – The time and space complexity (expressed in big-O notation) of the different evaluation techniques for the various computational models under rank-1 updates to matrix A where $2 \leq \gamma \leq 3$, as described in Section 6.2.

The performance improvement comes at the cost of increased memory consumption, as incremental maintenance requires storing the result of every iteration step. Table 6.2 also shows the space complexity of incremental maintenance for the three iterative models.

Sums of Matrix Powers

A form of matrix powers that frequently occurs in iterative computations is a sum of matrix powers. The goal is to compute $S_k = I + A + \ldots A^{k-2} + A^{k-1}$, for a given matrix A and fixed $k > 0$. Here, I is the identity matrix.

In Table 6.1, we express this computation using the iterative models discussed earlier. In the exponential and skip models, the computation of S_k relies on the results of matrix power computation, denoted by P_i and evaluated using the exponential model discussed earlier.

For all three models, the time and space complexity of computing sums of matrix powers is the same in terms of big-O notation as that of computing matrix powers. The intuition behind this result is that the complexity of each iteration step has remained unchanged. Each iteration step performs one matrix addition more, but the execution cost is still dominated by the matrix multiplication.

6.4.3 General Form: $T_{i+1} = A\,T_i + B$

The two examples of matrix power computation provide the basis for the discussion about a more general form of iterative computation: $T_{i+1} = A\,T_i + B$, where A and B are input matrices. In contrast to the previous analysis of matrix powers, this iterative computation involves also non-square matrices, $T = (n \times p)$, $A = (n \times n)$, and $B = (n \times p)$, making the choice of the optimum evaluation strategy dependent on the values of n, p, k, and s.

Many iterative algorithms share this form of computation including gradient descent, PageRank, iterative methods for solving systems of linear equations, and the power iteration method for eigenvalue computation. Here, we analyze the complexity of the general form of iterative computation, and the same conclusions hold in all these cases.

Iterative Models

The iterative models of the form $T_{i+1} = A\,T_i + B$, which are presented in Table 6.1, rely on the computations of matrix powers and sums of matrix powers. To understand the relationship between these computations, consider the iterative process $T_{i+1} = A\,T_i + B$ that has been "unrolled" for k iteration steps. The direct formula for computing T_{i+k} from T_i is $T_{i+k} = A^k\,T_i + (A^{k-1} + \ldots + A + I)\,B$.

We observe that A^k and $\sum_{i=0}^{k-1} A^i$ correspond to P_k and S_k in the earlier examples, for which we have already shown efficient (incremental) evaluation strategies. Thus, to compute T_i, we maintain two auxiliary views P_i and S_i evaluating matrix powers and sums of matrix powers using the exponential model discussed before.

Cost Analysis

We analyze the time and space complexity of re-evaluation and incremental maintenance of T_k for rank-1 updates to A, denoted by $\Delta A = u\,v^{\mathrm{T}}$. We assume that A is an $(n \times n)$ dense matrix and that T_i and B are $(n \times p)$ matrices. We can apply a similar analysis for changes in B.

We also analyze a combination of the two strategies, called hybrid evaluation, which avoids the factorization of delta expressions but instead represents them as single matrices. We consider this strategy because the size $(n \times p)$ of the delta matrix ΔT_i might be insufficient to justify the use of the factored form. For instance, consider an extreme case when T_i is a column vector $(p = 1)$, then ΔT_i has rank 1, and further decomposition into a product of two matrices would just increase the evaluation cost. In such cases, hybrid evaluation expresses ΔT_i as a single matrix and propagates it to the subsequent iterations.

Table 6.2 presents the time complexity of re-evaluation, incremental and hybrid evaluation of the $T_{i+1} = A\,T_i + B$ computation expressed in different iterative models for rank-1 updates to A. The same complexity results hold for the special form of iterative computation where $B = 0$. We discuss the results for each evaluation strategy next.

Table 6.2 also shows the space complexity of the three iterative models when executed using different evaluation strategies. The re-evaluation strategy maintains the result of T_i, and if needed P_i and S_i, only for the current iteration; it also stores the input matrices A and B. In contrast, the incremental and hybrid evaluation strategies materialize the result of every iteration, thus the memory consumption depends on the number of performed iterations.

Re-evaluation The choice of the iterative model with the best asymptotic behavior depends on the value of parameters n, p, k, and s. The time complexities from Table 6.2 shows that the linear model incurs the lowest time complexity when $p \ll n$, otherwise the exponential model dominates the others in terms of the running time. We analyze the cost of each iterative model next. Note that the re-evaluation strategy first updates A by ΔA and then recomputes T_i, and if needed P_i and S_i, using the new value of A.

- *Linear model* The computation performs k iterations, where each one incurs the cost of $\mathcal{O}(pn^2)$, and thus the total cost is $\mathcal{O}(pn^2k)$.
- *Exponential model* Maintaining P_i and S_i takes $\mathcal{O}(n^\gamma)$ operations as discussed before, while recomputing T_i requires $\mathcal{O}(pn^2)$ operations. Overall, the re-evaluation cost is $\mathcal{O}((n^\gamma + pn^2)log k)$.
- *Skip model* The skip model combines the above models, which reflects on the cost analysis. Maintaining P_s and S_s takes $\mathcal{O}(n^\gamma \log s)$ operations as shown earlier, while recomputing T_i costs $\mathcal{O}(pn^2)$ per iteration. The total number of $(\log_2 s + \frac{k}{s})$ iterations yields the total cost of $\mathcal{O}(n^\gamma \log s + pn^2(\log s + \frac{k}{s}))$.

Incremental Maintenance Table 6.2 shows that incremental evaluation of T_k using the exponential model incurs the lowest time complexity among the three iterative models. It also outperforms complete re-evaluation when $p > n$, but the performance benefit diminishes as p becomes smaller than n. For the extreme case when $p = 1$, complete re-evaluation and incremental maintenance have the same asymptotic behavior, but in practice re-evaluation performs fewer operations as it avoids the overhead of computing and propagating the factored deltas. We show next how to combine the best of both worlds to lower the execution time when $p \ll n$.

Hybrid evaluation Hybrid evaluation departs from incremental maintenance in that it represents the change in the result of every iteration as a single matrix instead of an outer product of two vectors. The benefit of hybrid evaluation arises when the rank of $\Delta T_i = (n \times p)$ is not large enough to justify the use of the factored form; that is, when the dimension p or n is comparable with k.

Table 6.2 shows the time complexity of hybrid evaluation of the $T_{i+1} = A\,T_i + B$ computation expressed in different iterative models for rank-1 updates to A. For the extreme case when $p = 1$, the skip model performs $\mathcal{O}(\log s + \frac{k}{s} + s)$ matrix-vector multiplications. In comparison, re-evaluation and incremental maintenance perform $\mathcal{O}(k)$ such operations. So, the skip model of hybrid evaluation bears the promise of better performance for the given values of k and s.

6.5 System Overview

We have built the LINVIEW system that implements incremental maintenance of analytical queries written as (iterative) linear algebra programs. It is a compilation framework that

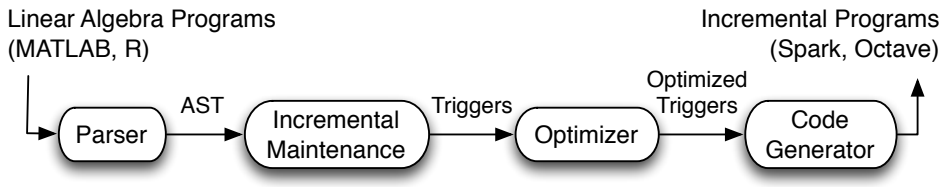


Figure 6.2 – The LINVIEW system overview

transforms a given program, based on the techniques discussed above, into efficient update triggers optimized for the execution on different runtime engines. Figure 6.2 gives an overview of the system.

Workflow The LINVIEW framework consists of several compilation stages: the system transforms the code written in APL-style languages (e.g., R, MATLAB, Octave) into an abstract syntax tree (AST), performs incremental compilation, optimizes produced update triggers, and generates efficient code for execution on single-node (e.g., MATLAB) or parallel processing platforms (e.g., Spark, Mahout, Hadoop). The generated code consists of trigger functions for changes in each input matrix used in the original program.

The optimizer analyzes intra- and inter-statement dependencies in the input program and performs transformations, like common subexpression elimination and copy propagation [113], to reduce the overall maintenance cost. In this process, the optimizer might define a number of auxiliary materialized views that are maintained during runtime to support efficient processing of the trigger functions.

Extensibility The LINVIEW framework is also extensible: one may add new frontends to transform different input languages into AST or new backends that generate code for various execution environments. At the moment, LINVIEW supports generation of Octave programs that are optimized for execution in multiprocessor environments, as well as Spark code for execution on large-scale cluster platforms. The experimental section evaluates both backends.

Distributed Execution The competitive advantage of incremental computation over re-evaluation – reduced computation time – is even more pronounced in distributed environments. Generated incremental programs are amenable to distributed execution as they are composed of the standard matrix operations, for which many specialized tools offer scalable implementations, like ScaLAPACK, Intel MKL, and Mahout. In addition, by transforming expensive matrix operations to work with smaller datasets and representing changes in factored form, our incremental techniques also minimize the communication overhead as less data has to be shipped over the network.

Data Partitioning LINVIEW analyzes data flow dependencies and data access patterns in a generated incremental program to decide on a partitioning scheme that minimizes data movement. A frequently occurring expression in trigger programs is a multiplication of a large

matrix and a small delta matrix, typically performed in both directions (e.g., $A\Delta A$ and $\Delta A A$ in Example 6.1.1). To keep such computations strictly local, LINVIEW partitions large matrices both horizontally and vertically, that is, each node contains one block of rows and one block of columns of a given matrix. Although such a hybrid partitioning strategy doubles the memory consumption, it allows the system to avoid expensive reshuffling of large matrices, requiring only small delta vectors or low-rank matrices to be communicated.

6.6 Experiments

This section demonstrates the potential of LINVIEW over traditional re-evaluation techniques by comparing the average view refresh time for common data mining programs under a continuous stream of updates. We have built an APL-style frontend where users can provide their programs and annotate dynamic matrices. The LINVIEW backend consists of two code generators capable of producing Octave and Spark executable code, optimized for the execution in multiprocessor and distributed environments.

For both Spark and Octave backends, our results show that:

1. Incremental view maintenance outperforms traditional re-evaluation in almost all cases, validating the complexity results of Section 6.4;

2. The performance gap between re-evaluation and incremental computation increases with higher dimensions;

3. The hybrid evaluation strategy from Section 6.4.3, which combines re-evaluation and incremental computation, exhibits best performance when the input matrices are not large enough to justify the factored delta representation.

Experimental Setup To evaluate LINVIEW's performance using Octave, we run experiments on a 2.66GHz Intel Xeon with 2 × 6 cores, each with 2 hardware threads, 64GB of DDR3 RAM, and Mac OS X Lion 10.7.5. We execute the generated code using GNU Octave v3.6.4, an APL-style numerical computation framework, which relies on the ATLAS library for performing multi-threaded BLAS operations.

For large-scale experiments, we use an Amazon EC2 cluster with 26 compute-optimized instances (c3.8xlarge). Each instance has 32 virtual CPUs, each of them is a hardware hyperthread from a 2.8GHz Intel Xeon E5-2680v2 processor, 60GB of RAM, and 2 × 320GB SSD. The instances are placed inside a non-blocking 10 Gigabit Ethernet network.

We run our experiments on top of the Spark engine – an in-memory parallel processing framework for large-scale data analysis. We configure Spark to launch 4 workers on one EC2 instance – 100 workers in total, each with 8 virtual CPUs and 13.6GB of RAM – and one master node on a separate EC2 instance. The generated Spark code relies on Jblas for performing matrix operations in Java. The Jblas library is essentially a wrapper around the BLAS and

LAPACK routines. For the purpose of our experiments, we compiled Jblas with AMD Core Math Library v5.3.1 (ACML) – AMD's BLAS implementation optimized for high performance. Jblas uses the ACML native library only for $\mathcal{O}(n^\gamma)$ operations, like matrix multiplication.

We implement matrix multiplication on top of Spark using the simple parallel algorithm [73], and we partition input matrices in a 10×10 grid. For the scalability test we use square grids of smaller sizes. For incremental evaluation, which involves multiplication with low-rank matrices, we use the data partitioning scheme explained in Section 6.5; we split the data horizontally among all available nodes, then broadcast the smaller relation to perform local computations, and finally, concatenate the result at the master node. The Spark framework carries out the data shuffling among nodes.

Workload Our experiments consider dense random matrices up to (100K × 100K) in size, containing up to 10 billion entries. All matrices have double precision and are preconditioned appropriately for numerical stability. For incremental evaluation, we also precompute the initial values of all auxiliary views and preload these values before the actual computation. We generate a continuous random stream of rank-1 updates where each update affects one row of an input matrix. On every such change, we re-evaluate or incrementally maintain the final result. The reported values show the average view refresh time over 3 runs; the standard deviation was less than 5% in each experiment.

Notation Throughout the evaluation discussion we use the following notation: (a) The prefixes REEVAL, INCR, and HYBRID denote traditional re-evaluation, incremental processing, and hybrid computation, respectively; (b) The suffixes LIN, EXP, and SKIP-S represent the linear, exponential, and skip-s models, respectively. These evaluation models are described in detail in Section 6.4 and Table 6.2.

6.6.1 Ordinary Least Squares

We conduct a set of experiments to evaluate the statistical estimator β^* using OLS as defined in Section 6.4.1. Exceptionally in this example, we present only the Octave results as the current Spark backend lacks the support for re-evaluation of matrix inversion. The predictors matrix X has dimension $(n \times n)$ and the responses matrix Y is of dimension $(n \times p)$. Given a continuous stream of updates on X, Figure 6.3 compares the average execution time of re-evaluation REEVAL and incremental maintenance INCR of β^* with different sizes of n. We set $p = 1$ because this setting represents the lowest cost for REEVAL, as the cost is dominated by the matrix inversion re-evaluation $\mathcal{O}(n^\gamma)$. The graph illustrates the superiority of INCR over REEVAL in computing the OLS estimates. Notice the asymptotically different behavior of these two graphs – the performance gap between REEVAL and INCR increases with matrix size, from 3.56x for $n = 4,000$ to 11.45x for $n = 20,000$ – which is consistent with the complexity results from Section 6.4.1.

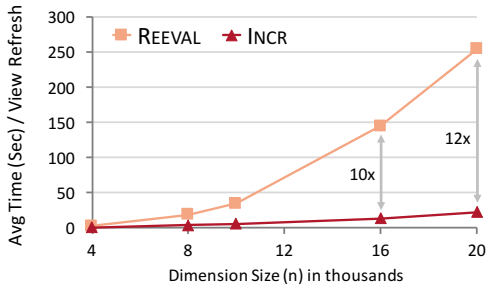


Figure 6.3 – Octave: Ordinary Least Squares $(X^T X)^{-1} X^T y$, where $X = (n \times n)$, $y = (n \times 1)$

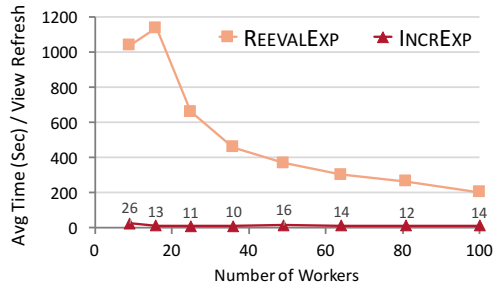


Figure 6.4 – Spark: Strong scalability of the A^{16} computation for $A = (30,000 \times 30,000)$

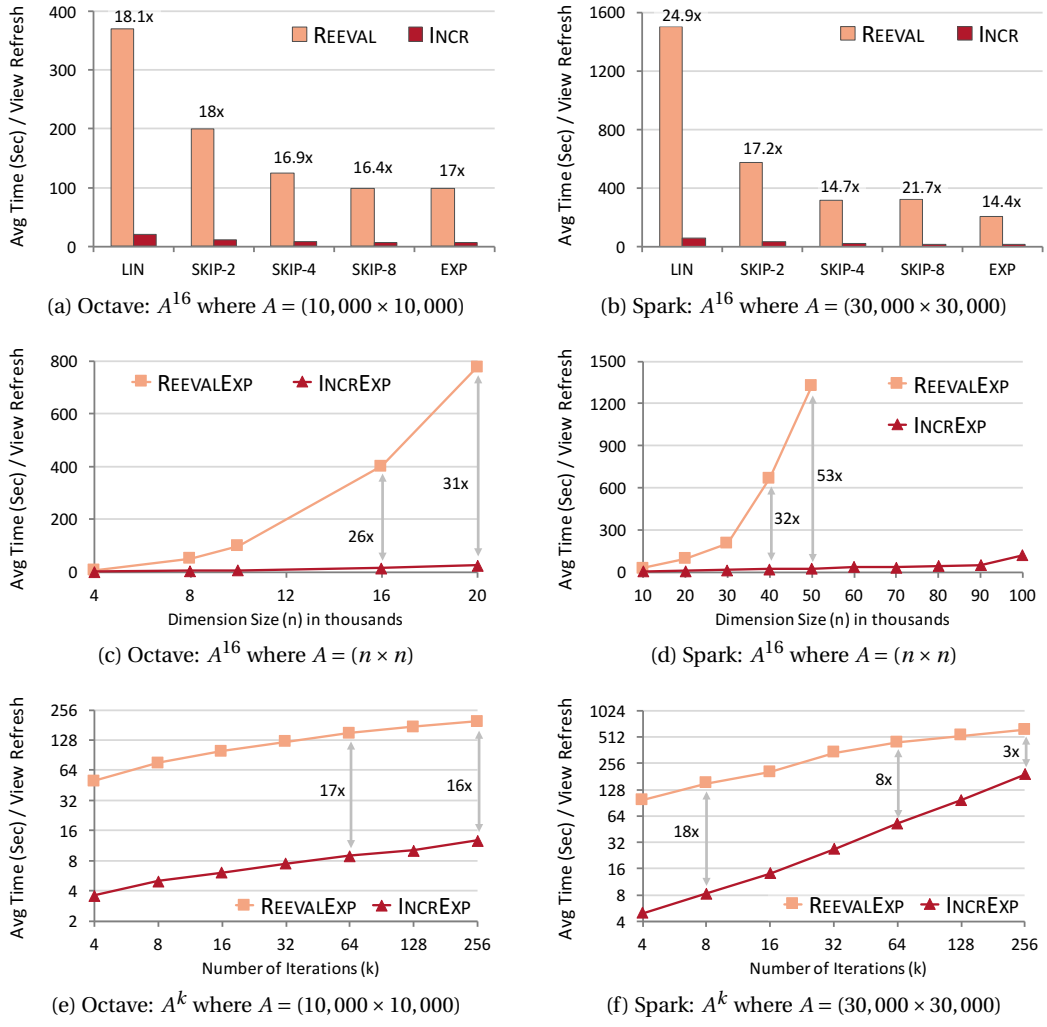
6.6.2 Matrix Powers

We analyze the performance of the matrix powers A^k evaluation, where A has dimension $(n \times n)$, by varying different parameters of the computational model. First, we evaluate the performance of the evaluation strategies presented in Section 6.4.2, for a fixed dimension size and number of iterations $k = 16$. Figures 6.5a and 6.5b illustrate the average view refresh time of Octave generated programs for $n = 10,000$ and Spark generated programs for $n = 30,000$. In both implementations, the results demonstrate the virtue of INCR over REEVAL and the efficiency of INCREXP over INCRLIN and INCRSKIP-S.

Next, we explore various scalability aspects of the matrix powers computation. Figure 6.5c reports on the Octave performance over larger dimension sizes n, given a fixed number of iteration steps $k = 16$; Figure 6.5d illustrates the Spark performance for even larger matrices. In both cases, INCREXP outperforms REEVALEXP with similar asymptotic behavior. As in the OLS example, the performance gap increases with higher dimensionality.

Figure 6.5d shows the Spark re-evaluation results for matrices up to size $n = 50,000$. Beyond this limit, the running time of re-evaluation exceeds one hour due to an increased communication cost and garbage collection time. The re-evaluation strategy has a more dynamic model of memory usage due to frequent allocation and deallocation of large memory chunks as the data gets shuffled among nodes. In contrast, incremental evaluation avoids expensive communication by sending over the network only relatively small matrices. Up until $n = 90,000$, we see a linear increase in the INCREXP running time. However, as discussed in Section 6.4, we expect the $\mathcal{O}(n^2)$ complexity for incremental evaluation. The explanation lies in that the generated Spark code distributes the matrix-vector computation among many nodes and, inside each node, over multiple available cores, effectively achieving linear scalability. For $n = 100,000$, incremental evaluation hits the resource limit in this cluster configuration, causing garbage collection to increase the average view refresh time.

Memory Requirements Table 6.3 presents the memory requirements and Spark single-update execution times of REEVALEXP and INCREXP for the A^{16} computation and various matrix dimensions. The last row represents the ratio between the speedup achieved using



(a) Octave: A^{16} where $A = (10,000 \times 10,000)$

(b) Spark: A^{16} where $A = (30,000 \times 30,000)$

(c) Octave: A^{16} where $A = (n \times n)$

(d) Spark: A^{16} where $A = (n \times n)$

(e) Octave: A^k where $A = (10,000 \times 10,000)$

(f) Spark: A^k where $A = (30,000 \times 30,000)$

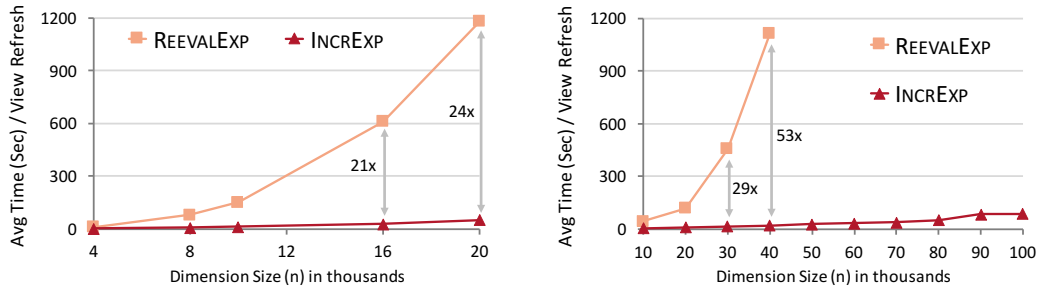
Figure 6.5 – Computing matrix powers (A^k) using Octave and Spark

incremental evaluation and the memory overhead imposed by maintaining the results of intermediate iterations. We conclude that the benefit of investing more memory resources increases with higher dimensionality of the computation.

Next, in Figures 6.5e and 6.5f, we vary the number of iteration steps k given a fixed dimension, $n = 10,000$ for Octave and $n = 30,000$ for Spark. The Octave performance gap between INCREXP and REEVALEXP increases with more iterations up to $k = 256$ when the size of the delta vectors ($10,000 \times 256$) becomes comparable with the matrix size. The Spark implementation broadcasts these delta vectors to each worker, so the achieved speedups decrease with larger iteration numbers due to the increased communication costs. However, as argued in Section 6.4.2, many iterative algorithms in practice require only a few iterations to converge, and for those the communication costs stay low.

Matrix Size		20K	30K	40K	50K
Memory	REEVALEXP	8.9	20.1	35.8	55.9
(GB)	INCREXP	29.8	67.1	119.2	186.3
Time	REEVALEXP	95.0	203.4	667.3	1328.7
(sec)	INCREXP	9.6	14.1	21.0	24.9
Speedup vs. Memory Cost		2.99	4.31	9.55	16.00

Table 6.3 – The memory requirements and Spark view refresh times of REEVALEXP and INCREXP for A^{16} and different matrix sizes. The last row is the ratio between the speedup and memory overhead incurred by maintaining auxiliary views.



(a) Octave: $A^0 + A^1 + \ldots + A^{15}$ where $A = (n \times n)$

(b) Spark: $A^0 + A^1 + \ldots + A^{15}$ where $A = (n \times n)$

Figure 6.6 – Computing a sum of matrix powers ($A^0 + A^1 + \ldots + A^{15}$) using Octave and Spark

Finally, we evaluate the strong scalability of the matrix powers computation for different numbers of Spark nodes. We evaluate various square grid configurations for re-evaluation of A^{16}, where $n = 30,000$[4]. Figure 6.4 shows that our Spark implementation of matrix multiplication scales with more nodes. Note that incremental evaluation is less susceptible to the number of nodes than re-evaluation; the average time per view refresh varies from 10 to 26 seconds.

6.6.3 Sums of Powers

We analyze the computation of sums of matrix powers, as described in Section 6.4.2. Since it shares the same complexity as the matrix powers computation, we present only the performance of the exponential models. Figure 6.6 compares INCREXP and REEVALEXP on various dimension sizes n using Octave and Spark, for a given fixed number of iterations $k = 16$. Similarly to the matrix powers results from Figures 6.5c and 6.5d, INCREXP outperforms traditional REEVALEXP, and the achieved speedup increases with n. Beyond $n = 40,000$, the Spark re-evaluation exceeds the one-hour time limit.

[4]To achieve perfect load balance with different grid configurations, we choose the matrix size to be the closest number to 30,000 that is divisible by the total number of workers.

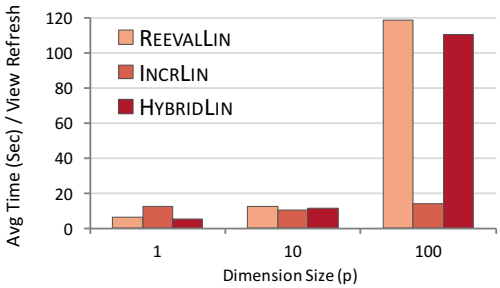


Figure 6.7 – Spark: General model $T_{i+1} = A\,T_i$ for $k = 16$ iterations, $A = (n \times n)$, $T = (n \times p)$, $n = 30,000$

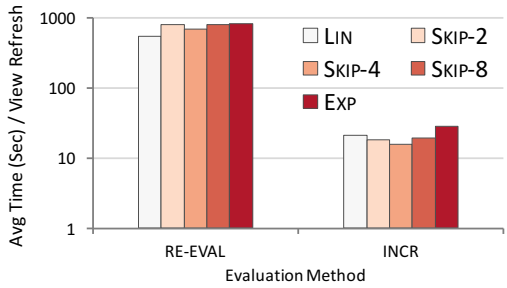


Figure 6.8 – Spark: Linear regression $T_{i+1} = A\,T_i + B$ for $k = 16$ iterations, $A = (n \times n)$, $T, B = (n \times 1,000)$, $n = 30,000$

6.6.4 General Form

We evaluate the general iterative model of computation $T_{i+1} = A\,T_i + B$, where $T_{n\times p}$, $A_{n\times n}$, and $B_{n\times p}$, using the following settings:

Case $\mathbf{B = 0}$ The iterative computation degenerates to $T_{i+1} = A\,T_i$, which represents matrix powers when $p = n$, and thus we explore an alternative setting of $1 \leq p < n$. For small values of p, the LIN model has the lowest complexity as it avoids expensive $\mathcal{O}(n^\gamma)$ matrix multiplications. Figure 6.7 shows the results of different evaluation strategies, given a fixed dimension $n = 30,000$ and iteration steps $k = 16$. For $p = 1$, HYBRIDLIN outperforms REEVALLIN by 16% and INCRLIN by 53%. However, the evaluation cost of both HYBRIDLIN and REEVALLIN increases linearly with p. INCRLIN exhibits the best performance among them when p is large enough to justify the factored delta representation.

Case $\mathbf{B \neq 0}$ We study an analytical query evaluating linear regression using the gradient descent algorithm of the form $\Theta_{i+1} = \Theta_i - X^{\mathrm{T}}(X\Theta_i - Y)$. We adapt this form to the general iterative model by substituting $A = I - X^{\mathrm{T}}X$ and $B = X^{\mathrm{T}}Y$, where I represents the identity matrix. Figure 6.8 shows the performance of different iterative models for both re-evaluation and incremental computation, given fixed sizes $n = 30,000$ and $p = 1,000$ and a fixed number of iterations $k = 16$. Note the logarithmic scale on the y axis. The LIN model exhibits the best re-evaluation performance; the SKIP-4 model has the lowest view refresh time for incremental evaluation. Overall, incremental computation outperforms traditional re-evaluation by a factor of 36.7x.

6.6.5 Batch updates

We analyze the performance of incremental matrix powers computation for batch updates. We simulate a use case in which certain regions of the input matrix are changed more frequently than the others, and the frequency of row updates is described using a Zipf distribution. Table 6.4 shows the performance of incremental evaluation for a batch of 1,000 updates and different Zipf factors. As the row update frequency becomes more uniform, that is, more rows

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

are affected by a given batch, INCREXP loses its advantage over REEVALEXP because the delta matrices become larger and more expensive to compute and distribute. To put these results in the context, a single update of a $n = 10,000$ matrix in Octave takes 99.1 and 6.3 seconds on average for REEVALEXP and INCREXP; For one update of a $n = 30,000$ matrix using Spark, REEVALEXP and INCREXP take 203.4 and 14.1 seconds on average. The Spark implementation incurs huge communication overheads, which significantly prolong the running time.

6.7 Summary

In this chapter, we studied the incremental view maintenance problem for complex analytical queries expressed as linear algebra programs. We have developed a framework, called LINVIEW, for capturing deltas of linear algebra programs and understanding their computational cost. Linear algebra operations tend to cause an avalanche effect where even very local changes to the input matrices spread out and infect all of the intermediate results and the final view, causing incremental view maintenance to lose its performance benefit over re-evaluation. We have developed techniques based on matrix factorizations to contain such epidemics of change. As a consequence, our techniques make incremental view maintenance of linear algebra practical and usually substantially cheaper than re-evaluation. We showed, both analytically and experimentally, the usefulness of these techniques when applied to standard analytics tasks. Our evaluation demonstrated the efficiency of LINVIEW in generating parallel incremental programs that outperform re-evaluation techniques by orders of magnitude.

7 Enabling Digital Signal Processing over Data Streams

An increasing proportion of today's data comes from networks of sensors and devices, commonly known as the Internet of Things (IoT). IoT workflows and applications typically run the same logic over a large collection of sensor devices using queries that combine relational and signal processing operations. Data analysts use relational operators, for example, to group signal by different sources or join signals with historical and reference data. They also use domain-specific algorithms such as Fast Fourier Transform (FFT) to do spectral analysis, interpolation to handle missing values, or digital filters to recover noisy signals. Reconciling these two seemingly disparate worlds, especially in the context of real-time analysis, is challenging.

The database community has recognized the need for a tighter integration of data management systems and domain-specific algorithms. Numerical computing environments like MATLAB and R provide efficient domain-specific algorithms but remain unsuitable for general-purpose processing involving relational operations such as joins, filtering, or group-by aggregation. To enable the use of specialized routines in complex data processing, increasingly many data management systems integrate with numerical frameworks, R in particular: MonetDB and SQL Server support queries that can invoke R code, which brings all the power of R packages inside a database; SciDB [139, 58, 41], Spark [152, 151], and Pivotal (Greenplum) [147] provide R packages that allow the user to interact with these systems directly from R. All these integrations benefit from re-using unmodified MATLAB/R scripts.

However, the existing integration mechanisms between database systems and numerical frameworks are suboptimal performance-wise as they treat both sides as independent systems. Such *loose system coupling* comes with significant processing overheads. For instance, executing R programs requires exporting data from the database, converting into R format, running R scripts, converting back into a relational format, and importing into the database. Sending data back and forth between the systems might dominate the execution time, for example when running (sub)linear R operators; it also increases the latency of processing, which makes this approach particularly unsuitable for real-time processing.

In this chapter, we advocate a *deep integration* of digital signal processing (DSP) operations with a general-purpose query processor. Our approach aims to bring signal processing closer to data, not the other way around and eliminate the need for expensive communication with external numerical tools. Integrating DSP operations into a query engine empowers users to express end-to-end workflows more succinctly, inside one system and using one language.

7.1 Challenges and Contributions

This tight integration poses several requirements and challenges:

1. *Query and data model reconciliation.* General-purpose query engines and numerical tools use different query and data models. The former support relational and streaming queries over typically relational or tempo-relational data; the latter support domain-specific, mostly offline, computations on arrays. The key challenge is how to seamlessly unify these disparate models instead of layering them on top of each other, and yet provide experts from both relational and signal processing worlds with familiar abstractions.

2. *Performance.* High performance is always a critical requirement for analytics. The deep integration approach brings more expressiveness to the query language but also carries a risk of throwing the baby out with the bathwater, that is, completely giving up on performance. Previous results suggested that simulating array computations on top of a relational database can yield orders of magnitude worse performance, which has motivated the development of array-based database systems [139, 58, 41]. In order to be welcomed by data scientists and DSP experts, a deeply integrated system should preserve the performance of existing relational operators while being competitive with MATLAB and R on pure domain-specific tasks. For workflows mixing relational and DSP processing, a tightly-coupled system should capitalize the potential of having much better performance than the existing loosely-coupled alternatives. For achieving this goal, the key challenge is how to efficiently integrate DSP operators inside a query processor.

3. *Extensibility.* A query processor with DSP support should allow domain experts and practitioners to implement custom operators in a way that feels natural to them – by writing algorithms against arrays without worrying about the format of the underlying data. Exposing arrays to operator writers enables easy integration of existing highly optimized DSP algorithms, for instance, implementations exploiting SIMD operations on modern processors. The system should seamlessly integrate new operators with the existing query language.

4. *Online and incremental computation.* Stream processors loosely coupled with MATLAB or R cannot incrementalize signal processing tasks that operate over hopping (overlapping) windows of data. The stateless nature of the DSP routines in MATLAB and R leaves no choice for stream engines but to redundantly compute over overlapping subsets of the data. On the other hand, the deep integration approach opens up the

opportunity for incremental DSP computation through the use of stateful operators. The user writing these operators should decide on which state to maintain and how to perform the incremental computation using the deltas provided by the system.

We extend TRILL [45], a query processing engine based on a tempo-relational model, with the support for digital signal processing. The deeply integrated system, named TRILLDSP, fulfills the above requirements: (1) it provides a unified query language for processing tempo-relational and signal data; (2) its performance is comparable to numerical tools like MATLAB and R and is orders of magnitude better than existing loosely-coupled data management systems; (3) it provides mechanisms for defining custom DSP operators and their integration with the query language; (4) it supports incremental computation in both offline and online analysis. The following example compares TRILLDSP against other commonly used systems.

7.1.1 Example: Spectral Analysis of Signals

An IoT application receives a stream of temperature readings from different sensors in time order. Each reading has the same format `<SensorId,Time,Value>`. The application runs the same query logic on every signal coming from a different source. The query consists of several processing stages, discussed next from the viewpoint of three different systems: R, SPARKR, and TRILLDSP. Table 7.1 shows their relevant code excerpts.

- *Stage 1: Grouping* The DSP routines in R cannot process multiple time-intertwined signals at once. Thus, the initial phase in R and SPARKR has to disentangle readings by their source (`SensorId`), while preserving the time order inside each group. The grouping operations in R (lines 1-2) and SPARKR (lines 1-10) are CPU and memory intensive tasks that involve copying the entire input. SPARKR consolidates the input data in parallel but requires local sorting of each group to restore the time order (line 9).

  TRILLDSP natively supports grouped processing through its data model and group-aware operators that internally maintain the state of each group. Grouping in TRILLDSP (line 2) is an in-place `Map` operation that associates a group identifier to each event in order to enable grouped processing in the downstream operators. Avoiding data copying brings orders of magnitude better performance than R and SPARKR, especially when handling large numbers of groups.

The workflow continues with processing each group either sequentially (R) or in parallel (SPARKR and TRILLDSP).

- *Stage 2: Interpolation* The values within one group might appear at irregular time intervals due to network delays or never appear due to message losses. To leverage DSP algorithms that mostly operate with equally-spaced sequences, the next processing stage transforms each group into a uniformly-sampled signal with the given sampling period and offset using linear interpolation. TRILLDSP hides from the user the burden of explicitly managing timestamps, in contrast to R (line 6) and SPARKR (line 13).

Listing 7.1 TRILLDSP

```
1  var q = stream.Map(s => s.Select(e => e.Value), e => e.SensorId)
2                .Reduce(s => s.Sample(100, 0, p => p.FirstOrder())
3                    .Window(512, 256, true,
4                        w => w.FFT().Select(a => f(a)).InverseFFT(),
5                        a => a.Sum()));
```

Listing 7.2 R

```
1  groups <- lapply(split(x,x[,1]), function(y) matrix(y,ncol=3)) # grp by id
2  z <- vector(''list'', length(groups))
3  for (x in groups) {
4    t <- x[, 2]; v <- x[, 3]                              # times and values
5    qt <- seq(t[1], t[length(t)], by = 100)                        # probes
6    y <- interp1(t, v, qt, method = 'linear')
7    frames <- window(y, 512, 256)                          # create frames
8    Y <- mvfft(frames)
9    Y2 <- f(Y)
10   y2 <- mvfft(Y2, inverse = TRUE)
11   z[[i]] <- unwindow(y2, 512, 256)                        # merge frames
12 }
```

Listing 7.3 SPARKR

```
1  grouped <- groupByKey(flatMap(rdd, function(x) {             # group by id
2    groups <- lapply(split(x,x[,1]), function(y) matrix(y,ncol=3))
3    lapply(groups, function(y) list(y[1, 1], y[, 2:3]))    # key-value pairs
4  }, cores)
5  sorted <- lapply(grouped, function(x) {                  # merge partitions
6    merged <- matrix(rbind(x[[2]]), ncol = 2)
7    merged[order(merged[, 1]),]                              # sort by time
8  })
9  result <- lapply(sorted, function(x) {
10   t <- x[, 1]; v <- x[, 2]                              # times and values
11   qt <- seq(t[1], t[length(t)], by = 100)                        # probes
12   y <- interp1(t, v, qt, method = 'linear')
13   frames <- window(y, 512, 256)                          # create frames
14   Y <- mvfft(frames)
15   Y2 <- f(Y)
16   y2 <- mvfft(Y2, inverse = TRUE)
17   unwindow(y2, 512, 256)                                  # merge frames
18 })
```

Figure 7.1 – Spectrum analysis in TRILLDSP, R, and SPARKR

Sampling and interpolation in TRILLDSP is a group-aware operator with full online support (line 3). The operator's ability to simultaneously process intertwined signals can result in up to 192x better performance than R and SPARKR, as seen in our experiments.

The remaining steps describe the fundamental technique in DSP [137]: (1) decompose the signal into simple components, (2) process each of the components in some useful way, and (3) recombine the processed components into the final signal.

- *Stage 3: Windowing* Most digital signal processing algorithms operate over windows of data defined by two parameters: the window size and the hop size. Using R (or SPARKR) misses the opportunity for incremental computation over hopping (overlapping) windows. Furthermore, invoking R routines for every window accumulates their startup costs. In the offline analysis, DSP experts often choose to copy windows out of the array and stack them into a matrix for batch processing. The `window()` function from

Table 7.1 forms such a matrix (details omitted for clarity) using twice as much memory for the given arguments.

The `Window` operator in TRILLDSP allows users to express a hopping window computation as a series of transformations of fixed-size arrays. One such pipeline transforms all input windows, thus amortizing the startup overhead, while supporting incremental computation. Based on the given window specification, the operator manages the data on behalf of users using circular buffers to avoid any redundancy in the input.

- *Stage 4: Spectral Analysis* The processing pipeline starts with the Fast Fourier Transform (FFT) that computes the frequency representation of 512-sample windows at each hopping point. A user-defined function f modifies the computed spectrum (e.g., retains top-k spectrum values with the highest magnitudes, zeros out others) before invoking the inverse FFT. The output stream has complex 512-size arrays at each hopping point.
- *Stage 5: Unwindowing* To restore the signal form, the final phase projects the output arrays back to the time axis and sums up the overlapping values. The `unwindow()` procedure from Table 7.1 carries out this task (7 lines omitted for clarity). In TRILLDSP, the framework performs this task using the provided aggregate function (line 6).

TRILLDSP has much lower code complexity than R and SPARKR, as evidenced in Table 7.1. The declarative query model of TRILLDSP allows expressing complex workflows using high-level operators; in contrast, R and SPARKR force users to write operations at a much lower level.

All three compared systems support offline analysis, but only TRILLDSP offers real-time capabilities. Also, note that the SPARKR program has hidden performance penalties as the consequence of loose coupling – each RDD function exports its input into R and imports the output back into Spark. Because of that and the other fundamental inefficiencies of loosely-coupled systems described above, SPARKR and SCIDB-R perform orders of magnitude worse than TRILLDSP in our experiments.

7.1.2 Contributions

To summarize, we make the following contributions:

1. Following the need for a deep integration of DSP operations and a general-purpose query processor, we provide a unified query and data model for relational and signal data. The model allows the end-user to seamlessly interleave tempo-relational and signal operations when writing relational and streaming queries, without ever explicitly dealing with array data.
2. For DSP experts, we provide frameworks for defining new user-defined window operations and their integration with the query language. The framework internally exposes array abstractions to ease the implementation for DSP experts and enables incremental computation with hopping windows.

Signal operations	Type	Ref
Relational operators (select, where, join, ALJ)	N+U	Trill
Arithmetic operations (+, -, *)	N+U	Trill
Basic signal operation (scale, shift)	N+U	Trill
Functional signal operations	N	§7.3.2
Sampling, upsampling, downsampling	N+U	§7.4.1
Interpolation	U	§7.4.2
Uniform signal aggregates (sum, power, energy)	U	Trill
Framework for user-defined digital filters (FIR & IIR filters, correlation, convolution)	U	§7.4.5
Framework for user-defined window operators (FFT, windowing functions, auto-correlation, cross-correlation, element-wise product)	U	§7.4.6

Table 7.1 – (N)on-uniform and (U)niform signal operations in TRILLDSP with references to the used operators or frameworks.

3. The unified query model supports both online and offline analysis. Users can build queries from offline data and then put them unmodified in production.

4. The performance of our system called TRILLDSP justifies the need for a deep integration DSP and query processing. On purely DSP tasks, TRILLDSP is comparable with best-of-breed for signal processing like MATLAB, Octave, and R. For queries mixing relational and signal processing, TRILLDSP shows up to 192x better performance than loosely-coupled systems like SPARKR [152, 151] and SCIDB-R [139, 58, 41].

Table 7.1 summarizes the supported signal operations in our system with the references to the operators or operator frameworks used in their implementations. Note that our system is extensible, so users can write custom operations using the provided frameworks.

This chapter is organized as follows. We provide a short overview of TRILL in Section 7.2. We introduce signal processing over data streams in Section 7.3 and then discuss processing of uniformly-sampled signals in Section 7.4. We present our experimental results in Section 7.5 and conclude in Section 7.6.

7.2 Background: The TRILL Library

TRILL [45] is a high-performance incremental analytics engine that uses the tempo-relational model and supports processing of streaming and relational queries. TRILL is written as a high-level language library (C#) that supports rich data-types and user libraries, and integrates well with existing distribution fabrics and applications.

7.2.1 Data Model

Trill represents stream data using the tempo-relational data model. Logically, we view a data stream consisting of events as a temporal database [86] that is presented incrementally [34, 80,

107]. Each stream event is associated with a data window (or an interval of application time) that denotes its period of validity. Such stream events form snapshots of valid data versions across time. The user query is executed against these snapshots in an incremental manner.

Trill represents a stream of data with payload type `T` as an instance of class `Streamable<T>`. In our introductory example, we can capture event contents using a C# payload type:

```
struct SensorReading {
  long SensorId;   long Time;   double Value;
}
```

The stream of sensor readings from our example is of type `Streamable<SensorReading>`.

`StreamEvent` is a TRILL structure that represents an event with payload type `T`. The structure provides static methods for creating stream events, including *point events* with a data window of one time unit. In our example, users may ingest sensor readings as point events at time t as:

```
StreamEvent.CreatePoint(t, new SensorReading { .. })
```

Physically, a dataset consists of a sequence of *columnar batches*. A columnar batch holds one array for each column in the event. For example, two arrays hold the start-time and end-time values for all events in the batch. TRILL associates a *grouping key* with each event in order to enable efficient grouped operations. TRILL precomputes and stores the grouping keys and key hash values as two additional arrays in a batch; it also includes an *absentee bitvector* to identify inactive rows in the batch.

TRILL creates columnar batches during query compilation. For example, the generated batch for `SensorReading` looks like:

```
class ColumnarBatchForSensorReading<TK> {
  long[] SyncTime;  long[] OtherTime;  // data window
  TK[] Key;         int[] Hash;
  long[] BitVector;
  long[] SensorId;  long[] Time;       // payload
  double[] Value;                      //
}
```

TRILL shares these arrays with reference counting; for example, `SyncTime` and `Time` may point to the same physical array. It also pools arrays using a global memory manager to alleviate the cost of memory allocation and garbage collection.

7.2.2 Query Language

TRILL's query language is modeled after LINQ [14], with temporal interpretation of the standard relational operations and new operations for temporal manipulation. These operations are exposed using the class `Streamable<T>`. Each TRILL operator is a function from stream to stream, which allows for elegant functional composition of queries. Each method represents a physical operator (e.g., `Where` for filtering) and returns a new `Streamable` instance, which

allows users to chain an entire query plan. For example, assuming `s0` is our data source of type `Streamable<SensorReading>`, we can discard invalid readings greater than 100 using the `Where` operator:

```
var s1 = s0.Where(e => e.Value > 100)
```

The expression in parentheses is called a lambda expression [6]; it is an *anonymous function*, in this case from `SensorReading` to `boolean` specifying the condition for keeping each stream event in the output stream `s1`.

An operator in TRILL accepts and produces a sequence of columnar batches. TRILL's compiler dynamically generate operators and inlines lambdas (such as the `Where` predicate) in tight per-batch loops to operate directly over columns for high performance. TRILL provides a rich set of built-in relational operators (e.g., selection, join, anti-join) as well as new temporal operators for defining windows and sessions.

Grouped computation TRILL extends the well-known Map and Reduce operations with temporal support to enable parallel query execution on each sub-stream corresponding to a distinct grouping key. Consider a shortened version of the query from Section 7.1.1:

```
var s2 =
  s1.Map(s => s.Select(e => e.Value), e => e.SensorId)
    .Reduce(s => s.Sample(10, 0),
      (k, p) => new Result { SensorId = k, Temp = p })
```

The first argument to `Map` specifies a sub-query (here, the stateless `Select` operation) to be performed in parallel on the input stream, while the second argument specifies the grouping key (`SensorId`) to be used for shuffling the result streams. The first argument to `Reduce` specifies the query to be executed per each group key (`Sample`), and the second argument allows us to combine the grouping key and the per-group payload into a single result.

Temporal join The temporal join operator in TRILL allows one to correlate (or join) two streams based on time overlap, with an (optional) equality predicate on payloads. Suppose we wish to augment the filtered `SensorReading` stream `s1` with additional information from another reference stream `ref1` that contains per-sensor location data. We would express such a query in TRILL as:

```
var s3 = s1.Join(ref1,
  l => l.SensorId, r => r.SensorId, (l,r) =>
  new Result { r.SensorLocation, l.Time, l.Value });
```

The second and third parameters to `Join` represent the equi-join predicate on the left and right inputs (`SensorId`), while the final parameter is a lambda expression that specifies how matching input tuples are combined to construct the result payload. The output stream `s3` is of type `Streamable<Result>`.

TRILL's query language is extensible in several ways. First, users can express user-defined aggregation logic by providing lambdas for accumulating and de-accumulating events to and

from state. Such logic is executed over columnar batches using a snapshot operator that maintains per-group state and inlines these lambdas in a tight loop [45]. Second, advanced users can write new operators that accept and produce a sequence of (grouped) columnar batches. Note that every TRILL operator understands grouping; for instance, the `Count` operator outputs a batched stream of per-key counts. Finally, every operator is transferable between real-time and offline by construction.

7.3 Signal Processing in TRILL

We consider signals as a special kind of streams in which stream events have no overlapping lifetimes. Representing signals in TRILL, thus, requires no changes of the underlying tempo-relational data model. The query model, however, needs to integrate new signal operators and enable their interleaving with the existing tempo-relational operators using one unified query language.

7.3.1 From Streams to Signals

Converting streams into signals needs to ensure that at most one stream event is active at any point in time. This property naturally emerges after applying an aggregate function over the input stream. For instance,

```
var s0 = stream.Average(e => e.Value)
```

creates a signal by averaging the overlapping stream events on the Value field. When the stream already has the signal form – for example, a sensor generates point events with lifetimes $[T, T+1)$ and increasing timestamps T– users can explicitly invoke `stream.ToSignal()` to get a signal. If the assertion turns out to be false later on, the system will throw a runtime error.

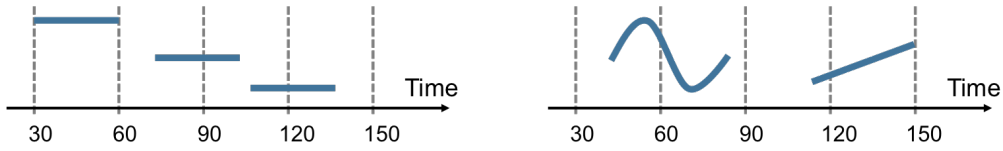
TRILLDSP considers streams having the signal form as instances of `SignalStreamable<T>`, which extends the base class `Streamable<T>` with signal operators. The system leverages the strong type-safety of C# to prevent invocation of signal operations over non-signal streams at compile time.

7.3.2 Signal Payload

Most often signals are discrete-time sequences of real or complex values called samples. Real-valued signals may originate from sensors measuring physical phenomena; complex-valued signals often emerge after processing signals in the frequency domain. Signals can also take more convoluted forms. For instance, in IoT applications, signals are often sequences of structures carrying multiple sensor measurements; in audio processing, signals comprise different audio channels; or in video processing, signals are sequences of movie frames. Handling these cases using existing signal processing tools requires a careful arrangement of these convoluted values into the matrix form before invoking operations on them.

```
1  interface TArithmetic<T> where T : TArithmetic<T> {
2      T Zero();                    T Plus(T b);
3      bool Equals(T b);            T Minus(T b);
4      T Scale(double scalar);      T Times(T b);
5  }
```

Figure 7.2 – Interface for defining custom signal payloads



(a) Constant payloads

(b) Functional payloads

Figure 7.3 – Signal payload types

TRILLDSP can process signals with arbitrary payloads, including real- and complex-valued signals, as long as the payload type `T` implements the interface `TArithmetic<T>`, shown in Figure 7.2. The interface consists of methods necessary for enabling signal processing on payloads of type `T`: a method defining a neutral element of `T`, an equivalence method, and four basic arithmetic operations over T (addition, subtraction, multiplication, and scalar multiplication). Using this interface, users can customize signal payloads for processing in different application domains.

Functional Signal Payloads

The tempo-relational data model defines stream events as having constant payloads over a time interval, as shown in Figure 7.3a. This model can naturally describe discrete-time signals and step-like continuous signals. In practice, users also want to work with more general continuous signals whose values can be expressed as a function of time, like in Figure 7.3b [1]. For example, in amplitude modulation, users multiply an input signal with a continuous carrier signal, which is usually a sine wave of a given frequency and amplitude.

To support (pseudo-)continuous signals with the tempo-relational data model, one can materialize point events at every timestamp from the function domain. Obviously, such an approach brings huge processing and memory overheads. To capture continuous signals efficiently, TRILLDSP introduces *functional signal payloads* that carry a lambda function describing how to compute signal values at any point in time. Functional payloads can delay materialization of signal events until the user explicitly asks for it. TRILLDSP supports two types of functional payloads, which we present next.

[1] Strictly speaking, continuous-time signals have a discrete nature due to the finite time resolution in TRILL.

Heterogeneous Functional Payloads

TRILLDSP allows users to assign a lambda function `Func<long,T>` to each stream event. The lambda function expresses the payload value as a function of time. TRILLDSP represents such functional payloads as instances of `TFuncPayload<T>`. For example, the user can create the first event from Figure 7.3b as:

```
var evt1 = StreamEvent.CreateInterval(40, 80,
  new TFuncPayload<double>(t => Sin(2 * Pi * t / 40)))
```

In order to enable processing of signals with functional payloads, `TFuncPayload<T>` implements `TArithmetic<TFuncPayload<T>>`. For example, adding two functional payloads creates a functional payload with a lambda function expressing the addition.

```
TFuncPayload<T> Plus(TFuncPayload<T> b) {
  return new TFuncPayload<T>(t => f(t) + b.f(t));
}
```

Processing signals with functional payloads is no different than processing any other payload type. For instance, adding two functional signals corresponds to a temporal join in which the matching events yield a summed functional payload as described above. Note that in processing signals with functional payloads, we always maintain the functional representation of values so as to avoid materialization and reduce processing and memory costs.

Users can explicitly materialize functional payloads at discrete time points to obtain a uniformly-sampled signal with the desired sampling period and offset. For example, one can sample a signal `fs0` containing functional payloads to materialize events at every 10 time units as:

```
var fs1 = fs0.Sample(10);          // Materialization
```

Using `TFuncPayload<T>` enables heterogeneity in assigning lambda functions to payloads, that is, each event can take a different function, like in Figure 7.3b. This function heterogeneity comes at the cost of more expensive memory management during query evaluation. New stream events need to instantiate lambda objects. Given that these lambdas have different structures, they cannot be efficiently shared among events or re-used through memory pooling. Instead, lambda objects are always allocated on the heap and released through garbage collection, which increases memory management overheads.

Homogeneous Functional Payloads

To process functional payloads more efficiently, TRILLDSP allows users to associate one lambda function to the entire signal rather than to each individual event. In that way, all stream events can share the same lambda function, hence "homogenous" in the name, and their payloads can represent the function arguments. For example, users can create stream events with payloads being the arguments of a sine function:

```
var event = StreamEvent.CreateInterval(0, 100,
  new SinArgs { Freq = 50.0, Phase = 0.0 });
```

Each signal maintains a property that stores its lambda function. Users can associate a sine function to the signal `s1` as follows:

```
var s2 = s1.setProperty().setFunction(
  (t,e) => Sin(2 * Pi * t / e.Freq + e.Phase))
```

The default signal function is the identity function `(t,e) => e`.

Supporting homogeneous functional payloads requires a change of the signature of the signal type from `SignalStreamable<T>` to `SignalStreamable<TIn,TOut>`, where `TIn` is the payload type (e.g., `SinArgs`) and `TOut` is the result type (e.g., `double`). The lambda function is of type `Func<long,TIn,TOut>`.

Binary signal operations over homogeneous functional payloads use temporal joins to pair matching payloads from both sides as 2-tuple structures. For example, multiplying the signal `s2` with a complex-valued constant signal yields payloads of `Tuple<SinArgs,Complex>`. Such payloads are arguments of the stream-level lambda function that multiplies these two signals.

Homogeneous functional payloads enable more efficient signal processing than heterogeneous functional payloads. During query compilation, we can flatten out homogeneous payloads into a sequence of parameters and create their columnar batch representation. In the previous example, the generated batch for the tuple structure looks like:

```
struct ColumnarTuple {
  double[] Freq;  double[] Phase;  Complex[] V;
}
```

These columnar arrays are memory-pooled and shared with reference counting during processing, which alleviates the cost of memory allocation and garbage collection. For performance reasons, TRILLDSP uses homogeneous functional payloads as the default payload type.

7.3.3 Signal Operators

The class `SignalStreamable<TIn,TOut>` encapsulates operators that preserve the signal form of a stream. It redefines such unary operators inherited from the base stream class, like `Where` and `Select`, to return a signal object. Since `Select` can change the input payload type `TIn`, users can also provide a new stream-level function for computing payload values or default to the identity function. The signal class also provides operators that build upon the existing stream operators. For example, `Scale` multiplies signal values with a scalar using the `Select` operator; `Shift` delays or advances a signal by changing the time intervals of its events using the alter-lifetime operator [45].

Users can perform basic arithmetic operations on signals, like addition, subtraction, or multiplication, as these operations are guaranteed to produce at most one event at any point in time. In fact, any signal operation that uses a temporal join to match events outputs a signal. Users just need to set the stream-level function to decide on how to combine matching events

into payload values. But not every binary stream operation can be lifted to the signal domain. For example, a union of two signals can produce a stream with overlapping events, so this operator needs to remain in the `Streamable` domain.

The signal type `SignalStreamable` also provides methods for obtaining uniformly-sampled signals, like the `Sample()` method that we used to materialize functional payloads. In the next section, we cover the operations over uniformly-sampled signals.

7.4 Uniformly-sampled Signals

Numerical frameworks like MATLAB and R support mostly DSP algorithms on signals consisting of equally-spaced samples. These tools consider such uniform signals as real- or complex-valued arrays in which the array index act as a measure of time.

TRILLDSP regards uniform signals as streams consisting of point events at regular timestamps. Uniform signals have two defining properties: (1) *sampling period* defines the time difference between two consecutive samples, and 2) *sampling offset* defines the initial time shift of samples. These two properties can help us correlate application times of samples with their positions in the array form.

TRILLDSP represents uniform signals as instances of `UniformSignalStreamable<T>`, which extends the signal class `SignalStreamable<_,T>` with uniform-signal operators. Uniform signals can have only point events at timestamps determined by their sampling period and offset. Regular (non-uniform) signals have no such constraints.

7.4.1 Sampling

We obtain a uniform signal by sampling a non-uniform signal. The `Sample` operator uses a given sampling period T and an offset O to generate point events at timestamps $kT + O$, where k is integer, at which the source signal has an active stream event. The `Sample` operator invokes the stream-level lambda function to compute the payload value of each materialized event. Since uniformly-sampled signals are discrete-time signals, they have no stream-level lambda function associated to them.

The `Sample` operator supports grouped signal computation. In grouped signals, each stream event carries a group identifier; the example from Table 7.1 groups stream events by `SensorId`. The `Sample` operator internally keeps track of active events for each group seen so far. Since TRILL ingests stream events in a non-decreasing sync (interval start) timestamp order, each event can move the global stream time forward. In such cases, the operator needs to produce samples for each group up to the current time, update the internal state of each group to remove any inactive events, and update the current group to include the current event. If the current state detects overlapping events, the operator throws a runtime error implying that user's assertion about the signal form of the input stream is false.

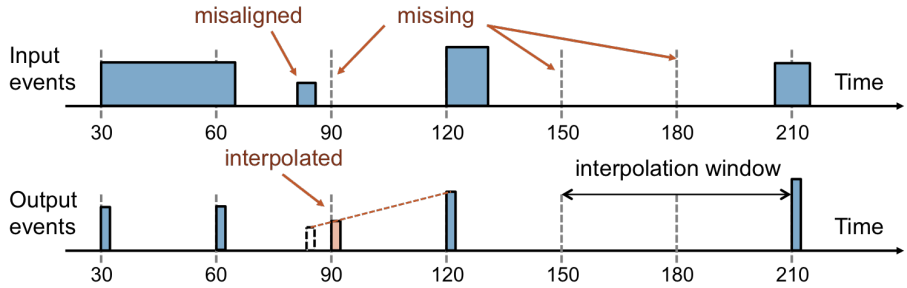


Figure 7.4 – Sampling with interpolation of a non-uniform signal

7.4.2 Interpolation

Non-uniform signals consist of overlapping-free stream events of arbitrary forms. As shown in Figure 7.4, a non-uniform signal can have events with different lifetimes, events appearing at irregular time instants (e.g., due to network delays), or missing events (e.g., due to bad communication). The `Sample` operator discussed so far considers only events that are active at predefined sampling beats (e.g., 30, 60, 90, etc.) and ignores in-between and missing events.

TRILLDSP supports sampling with interpolation of missing values. The `Sample` operator has an optional parameter – *interpolation policy* – that specifies the rules for computing missing events. An interpolation policy has two defining properties:

1. *Interpolation function* defines how to compute missing samples based on a fixed-size history of reference samples. TRILLDSP provides a set of common interpolation functions: constant, last-valued, step (zero-order), linear (first-order), and second-degree polynomial (second-order), and also supports user-defined interpolation functions.

2. *Interpolation window* defines the maximum time distance among the reference samples used for interpolation. For instance, in linear interpolation, if two consecutive reference samples are more than one interpolation window apart, we consider them as two different signal sequences and disable interpolation in that interval, like in Figure 7.4.

The user can obtain the uniform signal from Figure 7.4 as follows:

```
var s1 = s0.Sample(30, 0, p => p.FirstOrder(60));
```

Each uniformly-sampled signal maintains a property that stores an interpolation policy. This policy is used during signal re-sampling, which happens, for instance, in binary operations on uniform signals with different sampling periods.

Interpolation implementation

The `Sample` operator implements interpolation on grouped signals. For each group, the operator maintains a finite number of observed samples, which serve as reference points for a

given interpolation function. Each stream event with an interval $[a, b)$ generates a sequence of reference points: a start point at $[a, a+1)$, an end point at $[b-1, b)$, and all intermediate points at sampling beats from the interval (a, b).

A per-group interpolator state maintains most recent reference points using a fixed-size circular buffer. The buffer consists of at least two samples, which define the currently active interpolation window; higher-order interpolators may buffer more samples. Each interpolator state also encapsulates the interpolation function that corresponds to the chosen interpolation policy. The `Sample` operator interacts with each interpolator state independently of the interpolation function using the following methods:

- `AdvanceTime(long t)` invalidates points in the buffer that fall outside the interpolation window ending at time `t`.
- `AddPoint(long t, T v)` overwrites the oldest valid entry in the interpolation buffer with a given reference point.
- `CanInterpolate(long t)` returns true when the interpolation buffer is full and `t` is between the last two reference points.
- `Interpolate(long t)` calls the interpolation function to compute the value at time `t`.

As the global time moves forward with each stream event, the `Sample` operator updates the state of each group to include new and invalidate too distant reference points. When the interpolation buffer of one group is full, it interpolates events at the sampling beats between the last two reference points. Note that the operator interpolates events back in the past. To preserve the time order of output events, the `Sample` operator delays emitting output events by at most the size of the interpolation window.

7.4.3 Uniform-signal Operators

Resampling operations change the sampling rate of uniformly-sampled signals. `Upsample(n)` increases the sampling rate (i.e., decreases the sampling period) of a signal by an integer factor. The interpolation function computes the values of new intermediate samples. `Downsample(n)` decreases the sampling rate (i.e., increases the sampling period) of a signal by keeping every n-th sample and dropping the others. `Resample()` changes the sampling period, offset, and interpolation function of a signal. For instance, we can resample the uniform signal from Figure 7.4 to halve the sampling period, change the interpolation policy, and double the interpolation window size.

```
var s2 = s1.Resample(30, 0, ps => ps.ZeroOrder(120))
```

Binary arithmetic operations involving uniform signals rely on temporal joins for matching samples. For instance, summing two uniformly-sampled signals, written as `left.Plus(right)`, when they share the same sampling period and offset is implemented as:

```
left.Join(right, (l, r) => l + r)
```

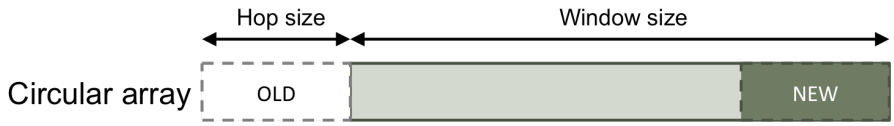


Figure 7.5 – Signal window

When these conditions are unmet, binary operators preprocess one of the operands before joining them together: if the sampling periods are the same but offsets are different, we shift one signal by the offset difference; if the sampling periods are different, we resample the signal with a larger sampling period to match the period and offset of the other signal; finally, when one operand is a non-uniform signal, we sample that signal with the sampling period and offset of the other uniform operand in order to produce a uniform result.

Uniform-signal aggregates combine existing stream operators. For instance, summing samples of a uniform signal `s` using a hopping window of size `W` and hop size `H` samples, written as `s.Sum(W,H)`, is implemented as:

```
s.HoppingWindow(W * s.Period, H * s.Period, s.Offset)
 .Aggregate(w => w.Sum(e => e))
 .AlterEventDuration(1)
```

where the `HoppingWindow` macro creates hopping windows by changing event lifetimes, the `Sum` aggregate adds values together, and the `AlterEventDuration` operator restores the uniform-signal form by emitting point events. Note that uniform-signal operators use sample-based windows. Uniform-signal aggregates can easily integrate any user-defined aggregate created using the extensible operator framework of TRILL [45].

7.4.4 Signal Windows

TRILLDSP represents uniformly-sampled signals as data streams built using a tempo-relational model. Numerical tools providing DSP algorithms like MATLAB and R consider uniformly-sampled signals as real- or complex-valued arrays, in which the array index is best thought of as a measure of dimensionless time. To bridge the gap between these two models, we introduce an abstraction that allows DSP users to implement custom operators for offline and online processing in a way that feels natural to them – by writing algorithms against arrays without worrying about the temporal aspect of the underlying data model. Exposing arrays to operator writers enables easy and quick integration of existing highly optimized DSP algorithms, like those exploiting SIMD operations on modern processors.

We introduce a class called `TWindow<T>` that transforms samples of type `T` from the tempo-relational data model into the array data model. In processing uniform signals using windows of data, a signal window maintains an array of active samples of one uniform signal. The signal window uses the sampling period and offset of the associated signal to compute the index position of a new sample. The signal window also keeps track of the latest timestamp in the

array to detect any missing value when adding a new sample; these missing values are either replaced with zeros or marked as invalid. Each signal window is implemented as a fixed-size circular array to efficiently support offline and online processing with overlapping windows.

Signal windows support incremental computation over hopping windows. Each signal window can maintain a fixed-size history of recently expired samples, as shown in Figure 7.5. The history size is often the window hop size. The `TWindow<T>` class provides methods for accessing the whole array (`Array`), the old delta (`Old`), and the new delta (`New`). Users can use these deltas to incrementally update operator states on every hop event.

Signal windows notify registered observers when their arrays are full. A signal window can fire up two types of events: 1) `OnInit` event denotes the array is filled up for the first time; the old delta is invalid, and the new delta is the whole array; 2) `OnHop` event denotes a window hop; Figure 7.5 shows both valid deltas.

TRILLDSP provides different implementations of signal windows based on their specification. A window specification comprises three parameters: the window size (how many samples each window lasts), the hop size (by how many samples each window moves forward relative to the previous one), and the boolean indicator on how to handle missing samples, if `true` replace them with zeros; otherwise, mark them as invalid. Note that signal windows containing invalid samples cannot fire up any event. The decision on how to treat missing values is operator-dependent. For instance, users might want to use complete arrays when using FFT but zero-padded signals when using digital filters.

7.4.5 Digital Filters

Digital filters are a cornerstone of digital signal processing. Due to their extraordinary efficiency, digital filters are widely used in practice; for example, filters can separate a signal from noise or restore bad audio recordings [137]. Digital filters are often described in terms of an equation that relates the output signal to the input signal. For example, linear digital filters produce outputs by combining a fixed-size window of inputs and a fixed-size window of previously computed outputs.

$$y[n] = \sum_{i=0}^{N} a[i] * x[n-i] + \sum_{i=1}^{M} b[i] * y[n-i]$$

Here, $x[i]$ and $y[i]$ are sequences of values denoting the input and output signals, $a[i]$ are called feed-forward coefficients, and $b[i]$ are called feed-backward coefficients. The feed-forward window is of size $N+1$, while the feed-backward window is of size M.

TRILLDSP provides a framework for defining custom digital filter. The framework uses `TWindow<T>` instances to represent feed-forward and feed-backward windows. Figure 7.6 shows an interface the user needs to implement in order to create a custom filter. The user

```
1  SetMissingDataToZero    () => bool
2  GetFeedForwardSize      () => int
3  GetFeedBackwardSize     () => int
4  Compute                 (TWindow<T>, TWindow<T>) => T
```

Figure 7.6 – Interface for defining custom digital filters

needs to specify the sizes of the feed-forward and feed-backward windows, decide on how to interpret missing values, and express the filtering functionality as a lambda function with a feed-forward and a feed-backward signal windows as parameters.

The digital filter framework supports grouped signal processing. A per-group state maintains two instances of `TWindow<T>`. The framework invokes the `Compute()` method upon receiving `OnInit` or `OnHop` events from the feed-forward window and updates the feed-backward window with the result; upon the `OnInit` event, it also resets the feed-backward window. Note that the framework releases users from the burden of explicitly managing windows in processing grouped signals and allows them to focus on the actual implementation.

An example of a digital filter is a finite impulse response filter, which uses only the feed-forward loop. For a given filter weights $a[i]$, each output is a weighted sum of the most recent inputs. Users can implement such filters as follows:

```
SetMissingDataToZero: () => true
GetFeedForwardSize:   () => a.Length
GetFeedBackwardSize:  () => 0
Compute:        (fw, bw) => DotProduct(a, fw.Array)
```

Signal windows expose array abstractions to users, which enables them to use highly-optimized black-box implementations of DSP algorithms. In this example, users can implement the `DotProduct` method using SIMD instructions of modern processors.

TRILLDSP uses the digital filter framework to implement several signal operators, like signal correlation, signal convolution, and finite and infinite impulse response filters to name a few.

7.4.6 User-defined Window Operators

DSP domain experts and practitioners exercise one fundamental workflow when analyzing uniform signals. The workflow consists of three steps [137]: (1) the first step splits the signal into smaller, possibly overlapping components, (2) the second step transforms each component using a sequence of operations, and (3) the third step recombines the processed components into the final signal.

TRILLDSP supports such workflows using the `Window` operator, which we present next using a shortened version of the query from Section 7.1.1 as our running example. We assume `s2` is a real-valued uniform signals.

```
1  GetOutputWindowSize     () => int
2  Update                  (TWindow<U>, TWindow<V>) => ()
3  Recompute               (TWindow<U>, TWindow<V>) => ()
```

Figure 7.7 – Interface for defining custom window pipeline operators

```
var s3 = s2.Window(512, 256, true,
  w => w.FFT().Select(a => f(a)).InverseFFT(),
  a => a.Sum()));
```

The first workflow step is about windowing the data. The `Window` operator transforms samples from streams into arrays using signal window instances of type `TWindow<T>`. To support grouped signal processing, the operator internally maintains one such instance for each group seen so far. Users can specify sample-based window attributes using the `Window` operator. As before, the window specification includes the window size, the hop size, and the policy on how to handle missing values (`true`: zero, `false`: NaN).

The second workflow step executes a sequence of transformations over fixed-size arrays. The `Window` operator expresses these computations as a pipeline of operators. Each pipeline operator transforms an input window of type `TWindow<U>` into an output window of type `TWindow<V>`. In our example, the `FFT` operator transforms a 512-sample window of type `TWindow<double>` into a 512-sample window of type `TWindow<Complex>` representing the frequency spectrum of the input windowed signal. In general, input and output windows can have different sizes; for instance, the `AutoCorrelation` pipeline operator takes windows of size N and outputs windows of size $2N+1$. The `Window` operator creates exactly one pipeline per signal group, which amortizes the startup overheads of pipeline operators; for instance, `FFT` and `InverseFFT` initialize their spectrum coefficients only once.

Each pipeline operator in TRILLDSP implements the interface presented in Figure 7.7. The first interface method defines the output window size of a pipeline operator. The `Window` operator uses this size to pre-allocate one output window for the pipeline operator. The other two methods define lambda functions for incremental computation and re-evaluation. Both methods change the output window in-place to avoid memory allocations. Note that operator writers can use black-box implementations for these operations. Also, it is their responsibility to enable incremental computation by denoting the delta parts of the output, if possible. Many signal operations, however, completely perturb their outputs making re-evaluation the only option for the downstream pipeline operators. In our example, only `FFT` can incrementally update its output; the remaining operators have to recompute their results from scratch. `InverseFFT` produces results in the output window of type `TWindow<Complex>`.

The final workflow step combines computed output windows into the final signal. In TRILLDSP, this step requires projecting output values from arrays back on the stream time axis as stream events. The `Window` operator projects N output values, which are generated at hop time `t`, backwards in time such that consecutive stream events are the sampling period apart from

each other and the last event has timestamp `t`. If the final output size is greater than the hop size, then the projected stream events overlap. To preserve the signal form of final stream, the `Window` operator allows users to provide an aggregate function (e.g., `Sum`) for merging overlapping output samples. When there are no overlapping events, we can safely lay out stream events on the stream time axis and yet preserve the signal form of the stream.

The last parameter of the `Window` operator is optional. If the aggregate function is omitted, the operator skips the "unwindowing" operation. Then, the output signal consists of output arrays as stream events defined at hop time points. In our example, if we omit the aggregate function, the output signal will be of type `UniformSignalStreamable<Complex[]>`. Similarly, when the final pipeline operator produces an output of type `T` that is different than `TWindow<_>`, then the output signal is of type `UniformSignalStreamable<T>`.

7.5 Performance Evaluation

In this section, we evaluate how TRILLDSP's deep integration of signal processing into a streaming query processor compares against state-of-the-art numerical frameworks specialized for signal processing and data analytics engines offering a loose integration with such frameworks. Our experiments aim to answer two main questions:

1. What is the price of integrating signal processing operations into a relational query processing system?
2. What is the benefit of having unified data and query models?

Our first set of experiments shows that a general-purpose query engine with tightly integrated signal processing operations can bring competitive or even better performance than popular numerical frameworks in pure signal processing tasks. These results come at no surprise as TRILLDSP has relatively low system overhead and exposes array abstractions to operator writers that enable quick and easy adoption of highly-optimized black-box implementations of DSP operations.

Our second set of experiments focuses on a particularly important class of IoT applications that run the same query logic over a large collection of sensor devices. Here, TRILLDSP shows its full potential when processing large numbers of groups using queries that combine relational and signal operations. TRILLDSP's in-situ execution model achieves from 3x to 192x better performance on grouped signal processing than modern relational and array data management systems with loose R integration.

Benchmarked systems We compare TRILLDSP against numerical frameworks that are widely used in practice by DSP experts and practitioners for offline signal analysis.

- MATLAB R2015b – a numerical computing environment offering a plethora of signal processing operations;

- Octave 4.0 – an open-source alternative to MATLAB;
- Revolution (Microsoft) R Open 3.2.3 (R for short) – an enhanced, open-sourced distribution of R used for statistical data analysis.

For workloads combining relational and signal processing, we compare TRILLDSP against the following loosely-coupled systems:

- Spark with R integration 1.5.2 (SPARKR) – an R package providing access to Spark from R;
- SciDB with R integration 14.12 (SCIDB-R) – a loosely-coupled integration.

The official integration of SciDB and R provides an R package for accessing the SciDB backend from R, same as Spark. However, this loosely-coupled approach suffers from huge overheads when serializing large datasets. For instance, just sending our dataset with 100 million events between SciDB and R without processing them takes more time than any total running time among the other compared system. We conclude that the official SCIDB-R integration is designed primarily for exchanging small aggregate results between these two systems. In order to make the SCIDB-R integration feasible on our datasets, we use a SciDB plugin [15] that enables running R programs within SciDB queries. In contrast to the official integration, here we write our queries as SciDB (not R) programs and run them from the SciDB (not R) shell.

Experimental Setup We run our experiments on a single node using D14 Microsoft Azure instances consisting of 2 Intel Xeon CPU E5-2673 v3 @ 2.40GHz and 112GB of RAM. We use one instance with Ubuntu 12.04.5 LTS for running Octave, R, SPARKR, and SCIDB-R queries, and another instance with Windows Server 2012 R2 Datacenter for running TRILLDSP and MATLAB queries.

Query Workload Our query workload includes five queries in total, which we evaluate using six different systems. Two queries represent pure DSP tasks commonly used in practice, while the remaining three queries combine relational and signal operations on grouped signals using Map-Reduce operators.

Data Workload Our datasets consist of 100 million randomly generated stream events. For the pure signal processing tasks, these events represent real-valued uniform signals with equally-spaced samples and the schema `<Time,Value>`. For the grouped computation queries, stream events also include the grouping key (`SensorId`) and have the schema `<SensorId,Time,Value>`. TRILLDSP never explicitly operates on the `Time` column as these values are implicitly encoded in the data windows of events.

7.5.1 Traditional DSP Tasks

We compare TRILLDSP with three numerical frameworks on signal processing tasks commonly used by DSP practitioners.

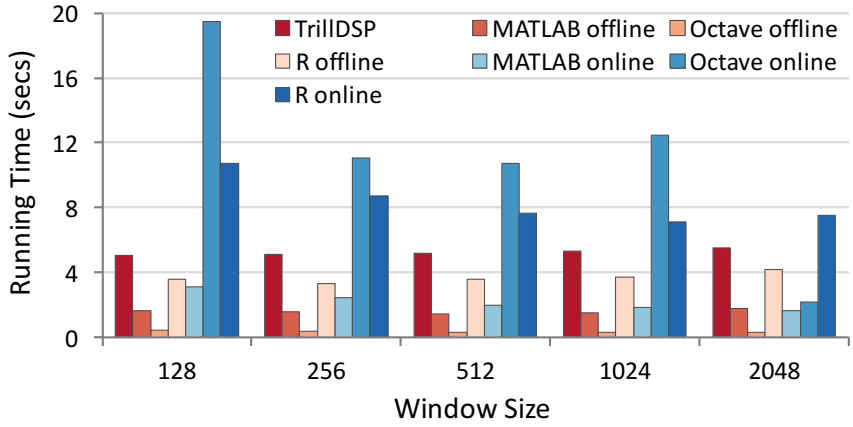


Figure 7.8 – FFT with tumbling window

FFT over tumbling windows

Our first query computes the Fast Fourier Transform (FFT) over tumbling windows of different sizes. In TRILLDSP, we write this query as:

```
var query1 = s0.Window(windowSize, hopSize, true,
    w => w.FFT(), a => a.Sum())
```

The input `s0` is a uniform signal; the output is a stream of complex values representing the frequency spectrum of each window.

For MATLAB, Octave, and R, we consider two different versions of programs: (1) the offline versions require the whole stream in order to repack signal values into a matrix form and then invoke a two-dimensional FFT only once; the reshaping technique is common among practitioners but obviously applicable only for offline analysis; (2) the online versions mimic a streaming environment in which FFT is called for each window.

Figure 7.8 presents the running times of these four systems when processing a dataset containing 100 million events. TRILLDSP manages to stay competitive with R offline, while outperforming R online and Octave online by 1.5-4x (except with the window size of 2048 when Octave evidently chooses a different FFT implementation). TRILLDSP is up to 3x slower than MATLAB in both online and offline mode and 15x slower than Octave offline.

FIR filtering Our next query uses a finite impulse response (FIR) filter to process a given uniform signal. The filter convolutes the signal with random filter coefficients. In TRILLDSP, this operation uses the digital filter framework presented in Section 7.4.5.

```
var query2 = s.FilterFIR(coeffs)
```

Figure 7.9 shows the filtering performance of our systems. Note the log scale on the y-axis. TRILLDSP consistently outperforms R and the performance gap increases with larger filter vectors, from 4x for 64-sized filters to 16x for 2048-sized filters. TRILLDSP stays competitive with Octave and is up to 1.8x slower than MATLAB.

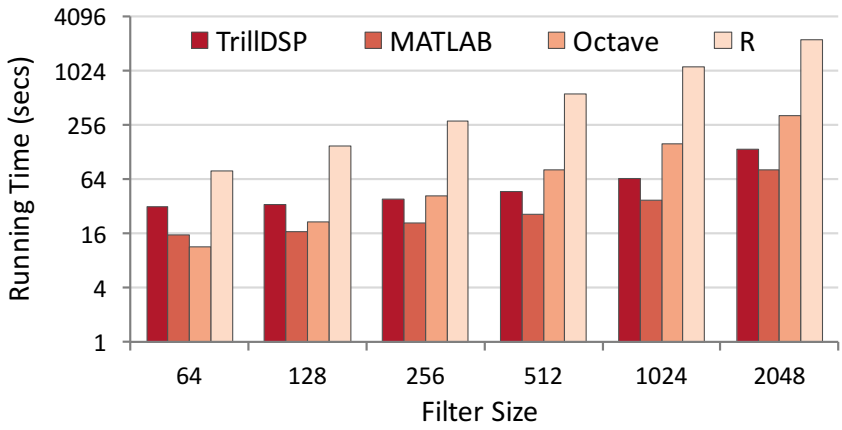


Figure 7.9 – Digital filtering

The goal of these experiments with standard DSP tasks is to demonstrate that TRILLDSP is competitive with or even better than the state-of-the-art tools used by practitioners. TRILLDSP's signal operations are efficient but probably not as well optimized as in the other systems, which exploit decades of research and algorithmic development in their implementations. Since TRILLDSP exposes arrays in its operator frameworks, users can easily integrate more sophisticated implementations, for instance, to exploit SIMD execution in FFT. But more importantly, TRILLDSP offers true online support and a tight integration with relational operators, which none of these systems does.

7.5.2 Grouped Signal Processing

Our next experiments involve grouped computation in which different sub-queries are executed for each group identified by the grouping key (`SensorId`). We consider three different sub-queries of growing complexity that combine signal processing and relational operations, and two new systems, SPARKR and SCIDB-R, capable of processing groups in parallel. We vary the number of groups while keeping the input signal size fixed.

Grouped Signal Correlation Our next query attempts to correlate a given vector of values of size 32, denoted as a, with a uniform signal of each group identified by `SensorId`. The correlation operation is a sliding window (i.e., the hop size is 1) computation that evaluates a dot product between the window and vector a at every sample point. In TRILLDSP, we express the query as:

```
var query3 =
  s0.Map(s => s.Select(e => e.Temp), e => e.SensorId)
    .Reduce(s => s.Correlation(a),
      (k, p) => new Result { SensorId = k, Temp = p })
```

The `Map` operation selects the value column and tags each stream event with its group identifier (`SensorId`), while the `Reduce` operation executes the sub-query on each group. TRILLDSP supports parallel execution of sub-queries using the specified level of parallelism. For these experiments, we evaluate TRILLDSP's performance using 1 and 16 cores.

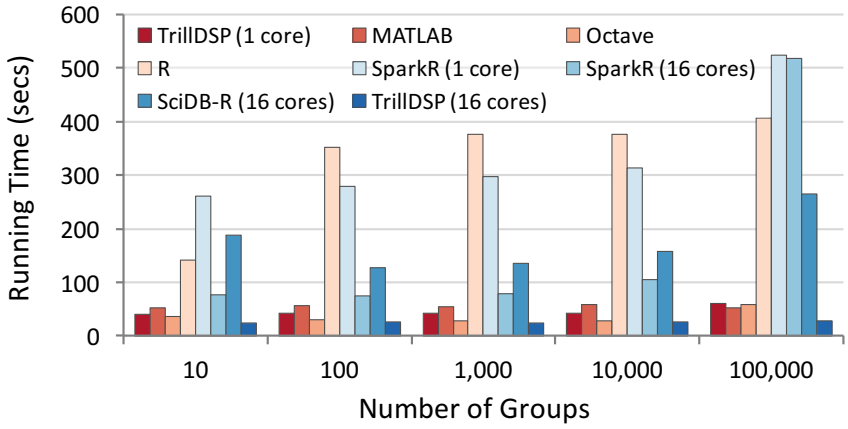


Figure 7.10 – Grouped signal correlation

The other systems implement the grouping operation differently. MATLAB and Octave partition the input signal using the `accumarray` method and then sequentially process each group in a for loop. Due to performance reasons, we partition the input signal in R using two methods, `accumarray` for groups of size 10 and `split` for larger groups. SPARKR uses `groupByKey` to partition an input RDD followed by a local sort of each partition, as shown in the example from Table 7.1. SCIDB-R uses the `redimension` operator to change the layout of array chunks based on the grouping key, also followed by a local sort of each group.

Figure 7.10 shows the query performance of these systems with different numbers of groups. TRILLDSP consistently outperforms all other systems by up to 20x. MATLAB, Octave, and R have expensive grouping operations that dominate over potentially efficient correlation operations, causing 2.2x, 1.4x, and 14x worse performance than TRILLDSP. SPARKR and SCIDB-R are slower than TRILLDSP by 3-20x, despite grouping and executing sub-queries in parallel. Note that both systems significantly degrade their performance when processing large numbers of groups.

Grouped signal interpolation and filtering Consider two uniformly-sampled signals `s1` and `s2` consisting of real-valued samples from $[0,1]$ and having different sampling periods, 20 and 5. For each group, we want to upsample the first signal to match the sampling period of the second and report events at which both group signals have simultaneously extreme values (close to 0 or 1). TRILLDSP expresses the grouping of `s1` as before (same for `s2`):

```
var u1 = s1.Map(s => s.Select(e => e.Temp), e => e.SensorId)
```

We write the query using a two-parameter `Reduce` operator as:

```
var query4 = u1.Reduce(u2, (l, r) =>
  l.Sample(5, 0, p => p.FirstOrder(20))
     .Plus(r).Where(e => e < 0.0001 || e > 1.9999),
  (k, p) => new Result { SensorId = k, Temp = p })
```

Figure 7.11 shows the query running times of different systems normalized by the running time of TrillDSP with 16 cores over varying numbers of groups. Note the log scale of the y-axis.

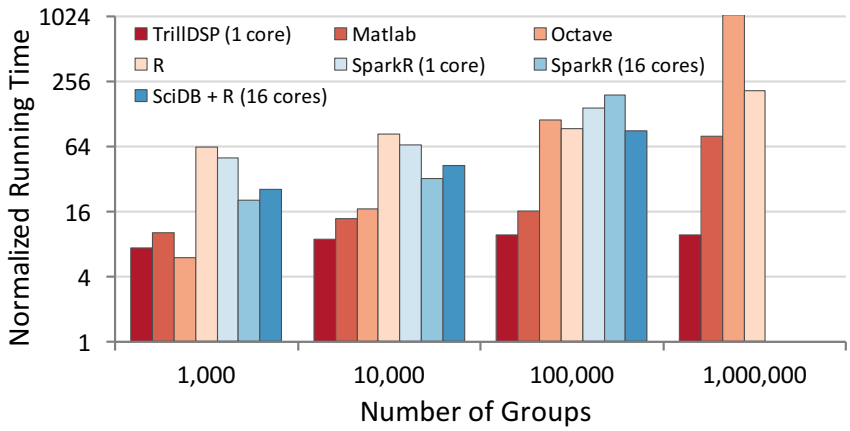


Figure 7.11 – Grouped signal interpolation and filtering

TRILLDSP supports stateful interpolation of grouped signals, which results in at least 10x better performance than the other systems and the performance gap increases with more groups. SPARKR and SCIDB-R crash when processing 1,000,000 groups, while Octave is three orders of magnitude slower than TRILLDSP. Note that TRILLDSP achieves around a 8x speedup using 16 cores during grouped processing.

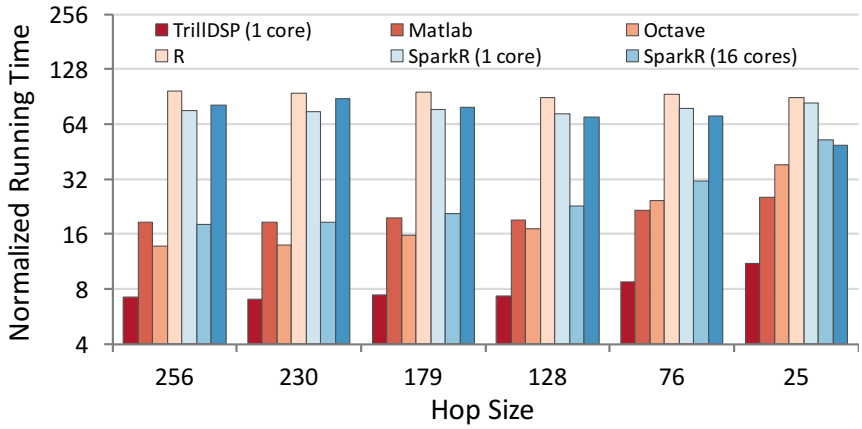


Figure 7.12 – Grouped overlap-and-add method

Grouped overlap-and-add Our final query implements one of the most important DSP techniques, the overlap-and-add method, on a uniformly-sampled grouped signal. The method follows the same steps described in Section 7.4.6. We modify the query from that section to use the identity function in the `Select` operator in order to measure the workflow performance independently of the user-defined function.

```
var query5 =
  s0.Map(s => s.Select(e => e.Temp), e => e.SensorId)
    .Reduce(s => s.Window(windowSize, hopSize, true,
        w => w.FFT().Select(w => w).InverseFFT(), a => a.Sum()),
      (k, p) => new Result { SensorId = k, Temp = p })
```

Figure 7.12 shows the overlap-and-add method performance in the benchmarked systems normalized by the performance of TRILLDSP with 16 cores over 256-sample hopping windows with different hop sizes. Note the log scale of the y-axis. The number of groups in the stream is 100. As with other grouped queries, TRILLDSP outperforms all other systems by up to two orders of magnitude. The reason this time lies in the use of circular arrays that effectively handle overlapping windows of data, especially when using small hop sizes.

7.6 Summary

In this chapter, we advocate for a deep integration of domain-specific tools and general-purpose query processors to support complex data analysis. Our approach unifies the seemingly disparate worlds of tempo-relational and array data and enables seamless interleaving of relational and signal operators within one query language, and yet provide experts from both relational and signal processing worlds with their familiar abstractions. Our system, called TRILLDSP, natively supports grouped signal processing in both online and offline mode, and provides extensible frameworks for domain experts to integrate new black-box operations.

TRILLDSP's performance is comparable with or better than the state-of-the-art numerical tools, like MATLAB and R. Our approach shows its full potential when processing large numbers of groups using queries that combine relational and signal operations. In these common IoT scenarios, TRILLDSP achieves from 3x to 192x better performance of grouped signal processing than modern data management systems with loose R integration.

8 Related Work

In this chapter, we provide a discussion of work related to the general themes in this thesis.

8.1 A Brief Survey of IVM Techniques

Database view management is a well-studied area with over three decades of supporting literature. Chirkova and Yang provide a survey of the topic [51]. We focus on the aspects of view materialization most pertinent to the DBToaster project. Our work innovates on the high-order aspect of incremental view maintenance, which is orthogonal to all previous work.

Traditional incremental view maintenance [76, 77, 75, 49, 51] is typically used for processing batch updates in data warehouse systems [22, 149, 128], where the focus is on achieving high throughput rather than low latency. Commercial database systems support incremental view maintenance but only for restricted classes of queries [9, 4].

Incremental View Maintenance Algorithms and Formal Semantics Maintaining query answers has been considered under both the set [38, 43] and bag [49, 74] relational algebra. Generally, given a query on N relations $Q(R_1,\ldots,R_N)$, traditional IVM uses a first-order delta query $\Delta_{R_1} Q = Q(R_1 \cup \Delta R_1, R_2, \ldots R_N) - Q(R_1,\ldots,R_N)$ for each input relation R_i in turn. The creation of delta queries has been studied for query languages with aggregation [123] and bag semantics [74], but we know of no work to formally examine delta queries of nested and correlated subqueries. View maintenance in nested relational algebra (NRA) [89, 96] has not been adopted in any commercial DBMS. Finally, incremental maintenance of temporal views [150] and views with outer joins and nulls [104] targets flat SPJAG queries without generalizing to subqueries, the full compositionality of SQL, or the range of standard aggregates.

Materialization and Query Optimization Strategies Selecting queries to materialize and reuse during processing has spanned fine-grained approaches from subqueries [131] and partial materialization [99, 132], to coarse-grained methods as part of multiquery optimization and common subexpressions [79, 159]. Picking views from a workload of queries typically uses the AND-OR graph representation from multiquery optimization [79, 131], or adopts signature

and subsumption methods for common subexpressions [159]. Previous work on selecting sets of subqueries of view definitions for materialization [131] is related to higher-order IVM as the delta of a simple query is frequently a subquery. Naturally, for such queries, both approaches select the same (optimal) materialization strategy. However, our delta operation also captures nonlinearities in the deltas of more complex queries (e.g., self-joins) and produces different materialization strategies. We also use query rewriting strategies that deal with correlated subqueries, inspired by work on query decorrelation [135]. Our experiments include results for a DBMS that uses a similar nested subquery materialization strategy. Additionally, our work builds on higher-order IVM, extending it into a complete query compilation system.

Physical DB designers [23, 160] often use the query optimizer as a subcomponent to manage the search space of equivalent views, reusing its rewriting and pruning mechanisms. For partial materialization methods, ViewCache [132] and DynaMat [99] use materialized view fragments, the former materializing join results by storing pointers back to input tuples, the latter subject to a caching policy based on refresh time and cache space overhead constraints.

Evaluation Strategies For efficient maintenance with first-order delta queries, previous work studies eager and lazy evaluation to balance query and update workloads [54, 158], asynchronous view maintenance [133], and the interaction of different view freshness models [55]. In addition, evaluating maintenance queries has been studied extensively in Datalog with semi-naive evaluation (which also uses first-order deltas) and DRed (delete-rederive) [77]. Finally, previous work on view maintenance in stream processing [70] reinforces our position of using IVM as a general-purpose change propagation mechanism for collections, on top of which window and pattern constructs can be defined.

8.2 Update Processing Mechanisms

Triggers and Active Databases Triggers, active databases, and event-condition-action (ECA) mechanisms [26] provide general purpose reactive behavior in a DBMS. The literature considers recursive and cascading trigger firings, and restrictions to ensure limited propagation. Trigger-based approaches require developers to manually convert queries to delta form, a painful and error-prone process especially in the higher-order setting. Without manual incrementalization, triggers suffer from poor performance and cannot be optimized by a DBMS when written in C or Java.

Data Stream Processing Data stream processing [18, 112] and streaming algorithms combine two aspects of handling updates: (1) shared, incremental processing (e.g., sliding windows) and (2) sublinear algorithms (with polylogarithmic space bounds). The latter are approximate processing techniques that are difficult to compose and have had limited adoption in commercial DBMS. Advanced processing techniques in the streaming community also focus mostly on approximate techniques when processing cannot keep up with stream rates (e.g., load shedding, prioritization [142]), on shared processing (e.g., on-the-fly aggregation [101]), or specialized algorithms and data structures [56]. Our approach to streaming is about generaliz-

ing incremental processing to (non-windowed) SQL semantics (including nested subqueries and aggregates). Of course, windows can be expressed in this semantics if desired. Similar principles are also discussed elsewhere [70].

Automatic Differentiation and Incrementalization Beyond databases, the programming language community has studied incremental computation [50] and automatic incrementalization [105]. It has developed languages and compilation techniques for translating high-level programs into executables that respond to dynamic changes. *Self-adjusting computation* targets incremental computation by exploiting dynamic dependency graphs and change propagation algorithms [19, 50]. These approaches: (a) serve for general purpose programs as opposed to our domain specific techniques, (b) require serious programmer involvement by annotating modifiable portions of the program, and (c) fail to efficiently capture deltas, as presented in this work. Automatic incrementalization is by no means a solved challenge, especially when considering general recursion and unbounded iteration. Automatic differentiation considers deltas of functions applied over scalars rather than sets or collections, and lately in higher-order fashion [125]. Bridging these two areas of research would be fruitful for supporting UDFs and general computation on scalars and collections in DBToaster.

8.3 Scalable Processing

Scalable Stream Processing Scalable stream processing platforms, such as MillWheel [28], Heron [102], and S4 [117], expose low-level primitives to the user for expressing complex query plans. S-Store [44] provides triggers for expressing data-driven processing with ACID guarantees in streaming scenarios. Naiad [114], Trill [45], and Dataflow [29] support declarative continuous queries but with no efficient support for the incremental computation of complex analytics, like queries with nested aggregates. Spark Streaming [153] allows running simple SQL queries but only for windowed data. In contrast to these systems, our approach: (1) favors declaritivity as the user only needs to specify input SQL queries without execution plans; (2) can incrementally maintain queries with equality-correlated nested aggregates; (3) generates code tailored to the given workload; our compilation framework can target any scalable system with a synchronous execution model. Percolator [126] handles incremental updates to large datasets but targets latencies on the order of minutes. Scalable batch processing systems like Pig, Hive, and Spark SQL [33] aim for high throughput rather than low latency. Their design favors infrequent bulk updates during which these systems are typically unresponsive.

Distributed Query Processing Our approach uses well-known techniques from distributed query processing [98, 67, 129], like the basic shuffling primitives and batch pre-processing for minimizing network communication. In contrast to classical distributed query optimization [20, 98, 47], optimizing higher-order incremental programs is more complex as one has to keep track of data-flow dependencies among program statements maintaining auxiliary views. These dependencies prevent arbitrary statement re-orderings inside trigger functions.

8.4 IVM in Linear Algebra

This thesis studies incremental computation of linear algebra programs, for which the challenges and optimization techniques differ widely from the previous work on incremental view maintenance of database (SQL) queries.

Iterative Computation Designing frameworks and computation models for iterative and incremental computation has received much attention lately. Differential dataflow [109] represents a new model of incremental computation for iterative algorithms, which relies on differences (deltas) being smaller and computationally cheaper than the inputs. This assumption does not hold for linear algebra programs because of the avalanche effect of input changes. Many systems optimize the MapReduce framework for iterative applications using techniques that cache and index loop-invariant data on local disks and persist materialized views between iterations [42, 66, 155]. More general systems support iterative computation and the DAG execution model,like Dryad [84] and Spark [152, 151]; Mahout [2], MLbase [100] and others [60, 145, 111] provide scalable machine learning and data mining tools. All these systems are orthogonal to our work. This thesis is concerned with the efficient re-evaluation of programs under incremental changes. Our framework, however, can be easily coupled with any of these underlying systems.

Scientific Databases There are also database systems specialized in array processing. RasDaMan [35], AML [108], and SciDB [139, 58, 41] provide support for expressing and optimizing queries over multidimensional arrays. ASAP [138] supports scientific computing primitives on a storage manager optimized for storing multidimensional arrays. RIOT [156] also provides an efficient *out-of-core* framework for scientific computing. However, none of these systems support incremental view maintenance for their workloads.

High Performance Computing The advent of numerical and scientific computing has fueled the demand for efficient matrix manipulation libraries. BLAS [64] provides low-level routines representing common linear algebra primitives to higher-level libraries, such as LINPACK, LAPACK, and ScaLAPACK for parallel processing. Hardware vendors such as Intel and AMD and code generators such as ATLAS [148] provide highly optimized BLAS implementations. In contrast, we focus on incremental maintenance of programs through efficient transformations and materialized views. The LINVIEW compiler translates expensive BLAS routines to cheaper ones and thus further facilitates adoption of the optimized implementations.

PageRank There is a huge body of literature that is focused on PageRank, including the Markov chain model, solution methods, sensitivity and conditioning, and the updating problem. Surveys can be found in [103, 37]. The updating problem studies the effect of perturbations on the Markov chain and PageRank models, including sensitivity analysis, approximation, and exact evaluation methods. In principle, these methods are particularly tailored for these specific models. In contrast, this work presents a novel model and framework for efficient incremental evaluation of *general* linear algebra programs through domain specific compiler translations and efficient code generation.

Incremental Statistical Frameworks Bayesian inference [36] uses Bayes' rule to update the probability estimate for a hypothesis as additional evidence is acquired. These frameworks support a variety of applications such as 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.

8.5 Signal Processing in Stream Engines

Signal Processing Numerical frameworks like MATLAB [10], R, and LabVIEW [8] provide a plethora of highly-optimized algorithms for signal processing but remain unsuitable for general-purpose stream and relational processing. Spark [152, 151] and SciDB [139, 58, 41] provide R packages that allow users to write R code that uses these systems as backend storages with scalable processing capabilities. MonetDB [39] and SQL Server [12] support queries that can invoke R routines. In contrast to all these systems, TRILLDSP uses one query language and one data model, performs in-situ processing of relational and signal data, and provides true online support. Plato [88] integrates signal processing models in a DBMS to deal with inaccurate or incomplete data but provides no support for general signal processing in both offline and streaming environments.

Stream Processing with DSP Functionality Conventional stream processing engines [32, 46, 18, 69] have no support for signal processing. StreamBase [13] enables integration with MATLAB and R but like other loosely-couple systems uses two different query languages and data models, requiring back and forth data exchanging. Gigascope [57] and Tribeca [140] are streaming database systems for networking applications that support relational queries but cannot express signal processing operations. StreamInsight [30] allows domain-specific extensions but provides no high-level signal processing abstractions (e.g., arrays) to the user. The XStream system [72], like our work, recognizes the fundamental need for implementing stream processing operations against an array abstraction (signal segments in XStream). This particular system, and its associated language, WaveScript [72, 71], exclusively target signal processing workloads and together provide a much lower level programming experience. For instance, arrays and their contents are visible and can be manipulated directly by the query writer outside the context of a user-defined operator.

Sensor networks TinyDB [106] and COUGAR [40] are sensor database systems supporting (tempo-)relational processing. Other specialized systems can handle sensor data, too [52]. But, none of these systems has native support for high-performance signal processing.

9 Conclusion

This research is motivated by the need for a class of systems optimized for keeping complex query analytics highly available and fresh under high update rates. In this thesis, we explore different application domains seeking solutions for the three key requirements of such systems: incremental processing, support for complex continuous queries, and scalable processing.

The DBToaster system has demonstrated the real power of combining novel query optimization and programming language techniques in building high-performance systems. Throughout the last several years, DBToaster has grown from a system that implements a naïve viewlet transform and interprets trigger statements to a system that supports distributed execution and exploits the state-of-the-art compiler libraries for aggressive code specialization. Compared to our first paper benchmarking the whole system [24], the performance numbers have gone up by additional $3-4$ orders of magnitude. DBToaster has attracted strong interest from both industry and academia, which makes the effort spent in building it well justified.

The second topic of the thesis, incremental linear algebra, teaches us how valuable theoretical results can be in the systems design. Our first Linview prototype was based on a complex algorithm for propagating delta expressions and maintaining their sparsity. The algorithm suffered from poor performance and was hard to parallelize. Discovering the matrix factorization idea has helped us to understand the research problem in a profound way, paving the way for a simpler, faster, and scalable implementation of the entire system.

The last part of the thesis demonstrates that one can extend general-purpose query processors with domain-specific functionality and get the best of both worlds: a more powerful, declarative query language and high performance of both standard and specialized operations. This result opens up the question whether the "one-size-doesn't-fit-all" paradigm from database systems [138] also holds for stream processing systems, in particular, because of their ability to transform data on-the-fly to meet different processing requirements. Answering this question is an interesting direction for future work.

A Appendix

A.1 Workload Queries

We list our TPC-H queries (with changes described in Section 4.3) and financial queries.

A.1.1 TPC-H Queries

Q1

```
SELECT returnflag, linestatus,
  SUM(quantity) AS sum_qty,
  SUM(extendedprice) AS sum_base_price,
  SUM(extendedprice * (1-discount)) AS sum_disc_price,
  SUM(extendedprice * (1-discount)*(1+tax)) AS sum_charge,
  AVG(quantity) AS avg_qty,
  AVG(extendedprice) AS avg_price,
  AVG(discount) AS avg_disc,
  COUNT(*) AS count_order
FROM Lineitem
WHERE shipdate <= DATE('1997-09-01')
GROUP BY returnflag, linestatus;
```

Q2

```
SELECT s.acctbal, s.name, n.name, p.partkey, p.mfgr,
       s.address, s.phone, s.comment
FROM Part p, Supplier s, Partsupp ps,
     Nation n, Region r
WHERE p.partkey = ps.partkey
AND   s.suppkey = ps.suppkey
AND   p.size = 15
AND   (p.type LIKE '%BRASS')
AND   s.nationkey = n.nationkey
AND   n.regionkey = r.regionkey
AND   r.name = 'EUROPE'
AND   (NOT EXISTS
            (SELECT 1
             FROM Partsupp ps2, Supplier s2,
                  Nation n2, Region r2
             WHERE p.partkey = ps2.partkey
             AND   s2.suppkey = ps2.suppkey
             AND   s2.nationkey = n2.nationkey
             AND   n2.regionkey = r2.regionkey
             AND   r2.name = 'EUROPE'
             AND   ps2.supplycost < ps.supplycost));
```

Q3

```
SELECT o.orderkey,
       o.orderdate,
       o.shippriority,
       SUM(extendedprice * (1 - discount)) AS query3
FROM   Customer c, Orders o, Lineitem l
WHERE  c.mktsegment = 'BUILDING'
AND    o.custkey = c.custkey
AND    l.orderkey = o.orderkey
AND    o.orderdate < DATE('1995-03-15')
AND    l.shipdate > DATE('1995-03-15')
GROUP BY o.orderkey, o.orderdate, o.shippriority;
```

Q4

```
SELECT o.orderpriority, COUNT(*) AS order_count
FROM   Orders o
WHERE  o.orderdate >= DATE('1993-07-01')
AND    o.orderdate <  DATE('1993-10-01')
AND   (EXISTS (
        SELECT * FROM Lineitem l
        WHERE l.orderkey = o.orderkey
        AND   l.commitdate < l.receiptdate
      ))
GROUP BY o.orderpriority;
```

Q5

```
SELECT n.name,
       SUM(l.extendedprice * (1 - l.discount)) AS revenue
FROM   Customer c, Orders o, Lineitem l, Supplier s,
       Nation n, Region r
WHERE  c.custkey = o.custkey
AND    l.orderkey = o.orderkey
AND    l.suppkey = s.suppkey
AND    c.nationkey = s.nationkey
AND    s.nationkey = n.nationkey
AND    n.regionkey = r.regionkey
AND    r.name = 'ASIA'
AND    o.orderdate >= DATE('1994-01-01')
AND    o.orderdate <  DATE('1995-01-01')
GROUP BY n.name;
```

Q6

```
SELECT SUM(l.extendedprice*l.discount) AS revenue
FROM   Lineitem l
WHERE  l.shipdate >= DATE('1994-01-01')
AND    l.shipdate < DATE('1995-01-01')
AND    (l.discount BETWEEN (0.06 - 0.01) AND (0.06 + 0.01))
AND    l.quantity < 24;
```

Q7

```
SELECT supp_nation, cust_nation, l_year,
       SUM(volume) AS revenue
FROM (
  SELECT n1.name AS supp_nation,
         n2.name AS cust_nation,
         EXTRACT(year from l.shipdate) AS l_year,
         l.extendedprice * (1 - l.discount) AS volume
  FROM   Supplier s, Lineitem l, Orders o, Customer c,
         Nation n1, Nation n2
  WHERE  s.suppkey = l.suppkey
  AND    o.orderkey = l.orderkey
  AND    c.custkey = o.custkey
  AND    s.nationkey = n1.nationkey
  AND    c.nationkey = n2.nationkey
  AND   ((n1.name = 'FRANCE' AND n2.name = 'GERMANY')
         OR
         (n1.name = 'GERMANY' AND n2.name = 'FRANCE'))
  AND (l.shipdate BETWEEN DATE('1995-01-01')
                      AND DATE('1996-12-31') )
  ) AS shipping
GROUP BY supp_nation, cust_nation, l_year;
```

Q8

```
SELECT  total.o_year,
        (SUM(CASE total.name WHEN 'BRAZIL'
                             THEN total.volume
                             ELSE 0 END)  /
         LISTMAX(1, SUM(total.volume)) ) AS mkt_share
FROM (
    SELECT n2.name,
           EXTRACT(year from o.orderdate) AS o_year,
           l.extendedprice * (1-l.discount) AS volume
    FROM   Part p, Supplier s, Lineitem l, Orders o,
           Customer c, Nation n1, Nation n2, Region r
    WHERE  p.partkey = l.partkey
    AND    s.suppkey = l.suppkey
    AND    l.orderkey = o.orderkey
    AND    o.custkey = c.custkey
    AND    c.nationkey = n1.nationkey
    AND    n1.regionkey = r.regionkey
    AND    r.name = 'AMERICA'
    AND    s.nationkey = n2.nationkey
    AND    (o.orderdate BETWEEN DATE('1995-01-01')
                            AND DATE('1996-12-31'))
    AND    p.type = 'ECONOMY ANODIZED STEEL'
  ) total
GROUP BY total.o_year;
```

Q9

```
SELECT nation, o_year, SUM(amount) AS sum_profit
FROM (
  SELECT n.name AS nation,
         EXTRACT(year from o.orderdate) AS o_year,
         ((l.extendedprice * (1 - l.discount)) -
          (ps.supplycost * l.quantity))  AS amount
  FROM   Part p, Supplier s, Lineitem l, Partsupp ps,
         Orders o, Nation n
  WHERE  s.suppkey = l.suppkey
  AND    ps.suppkey = l.suppkey
  AND    ps.partkey = l.partkey
  AND    p.partkey = l.partkey
  AND    o.orderkey = l.orderkey
  AND    s.nationkey = n.nationkey
  AND    (p.name LIKE '%green%')
  ) AS profit
GROUP BY nation, o_year;
```

Q10

```
SELECT c.custkey, c.name,
       c.acctbal,
       n.name,
       c.address,
       c.phone,
       c.comment,
       SUM(l.extendedprice * (1 - l.discount)) AS revenue
FROM   Customer c, Orders o, Lineitem l, Nation n
WHERE  c.custkey = o.custkey
AND    l.orderkey = o.orderkey
AND    o.orderdate >= DATE('1993-10-01')
AND    o.orderdate < DATE('1994-01-01')
AND    l.returnflag = 'R'
AND    c.nationkey = n.nationkey
GROUP BY c.custkey, c.name, c.acctbal, c.phone,
         n.name, c.address, c.comment;
```

Q11

```
SELECT p.partkey, SUM(p.value) AS QUERY11
FROM
  (
    SELECT ps.partkey,
           SUM(ps.supplycost * ps.availqty) AS value
    FROM   Partsupp ps, Supplier s, Nation n
    WHERE  ps.suppkey = s.suppkey
    AND    s.nationkey = n.nationkey
    AND    n.name = 'GERMANY'
    GROUP BY ps.partkey
  ) p
WHERE p.value > (
    SELECT SUM(ps.supplycost * ps.availqty) * 0.001
    FROM   Partsupp ps, Supplier s, Nation n
    WHERE  ps.suppkey = s.suppkey
    AND    s.nationkey = n.nationkey
    AND    n.name = 'GERMANY'
  )
GROUP BY p.partkey;
```

Q12

```
SELECT l.shipmode,
       SUM(CASE WHEN o.orderpriority IN ('1-URGENT',
                                         '2-HIGH')
                THEN 1 ELSE 0 END) AS high_line_count,
       SUM(CASE WHEN o.orderpriority NOT IN ('1-URGENT',
                                             '2-HIGH')
                THEN 1 ELSE 0 END) AS low_line_count
FROM   Orders o, Lineitem l
WHERE  o.orderkey = l.orderkey
AND   (l.shipmode IN ('MAIL', 'SHIP'))
AND    l.commitdate < l.receiptdate
AND    l.shipdate < l.commitdate
AND    l.receiptdate >= DATE('1994-01-01')
AND    l.receiptdate < DATE('1995-01-01')
GROUP BY l.shipmode;
```

Q13

```
SELECT c_count, COUNT(*) AS custdist
FROM (
   SELECT c.custkey AS c_custkey,
          COUNT(o.orderkey) AS c_count
   FROM Customer c, Orders o
   WHERE c.custkey = o.custkey
   AND   (o.comment NOT LIKE '%special%requests%')
   GROUP BY c.custkey
) c_orders
GROUP BY c_count;
```

Q14

```
SELECT (100.00 *
        SUM(CASE WHEN (p.type LIKE 'PROMO%')
                 THEN l.extendedprice * (1 - l.discount)
                 ELSE 0 END) /
            LISTMAX(1,
                    SUM(l.extendedprice * (1 - l.discount)))
        ) AS promo_revenue
FROM  Lineitem l, Part p
WHERE l.partkey = p.partkey
AND   l.shipdate >= DATE('1995-09-01')
AND   l.shipdate <  DATE('1995-10-01');
```

Q15

```
SELECT s.suppkey, s.name, s.address, s.phone,
       r1.total_revenue as total_revenue
FROM Supplier s,
     (SELECT l.suppkey AS supplier_no,
             SUM(l.extendedprice * (1 - l.discount))
              AS total_revenue
      FROM Lineitem l
      WHERE l.shipdate >= DATE('1996-01-01')
        AND l.shipdate <  DATE('1996-04-01')
      GROUP BY l.suppkey) r1
WHERE s.suppkey = r1.supplier_no
AND  (NOT EXISTS
          (SELECT 1
           FROM (SELECT l.suppkey,
                        SUM(l.extendedprice *
                            (1 - l.discount))
                         AS total_revenue
                 FROM   Lineitem l
                 WHERE  l.shipdate >= DATE('1996-01-01')
                 AND    l.shipdate <  DATE('1996-04-01')
                 GROUP BY l.suppkey) AS r2
           WHERE r2.total_revenue > r1.total_revenue) );
```

Q16

```
SELECT  p.brand, p.type, p.size,
        COUNT(DISTINCT ps.suppkey) AS supplier_cnt
FROM    Partsupp ps, Part p
WHERE   p.partkey = ps.partkey
AND     p.brand <> 'Brand#45'
AND     (p.type NOT LIKE 'MEDIUM POLISHED%')
AND     (p.size IN (49, 14, 23, 45, 19, 3, 36, 9))
AND     (ps.suppkey NOT IN (
            SELECT s.suppkey FROM Supplier s
            WHERE  s.comment LIKE '%Customer%Complaints%'))
GROUP BY p.brand, p.type, p.size;
```

Q17

```
SELECT SUM(l.extendedprice) / 7.0 AS avg_yearly
FROM   Lineitem l, Part p
WHERE  p.partkey = l.partkey
AND    p.brand = 'Brand#23'
AND    p.container = 'MED BOX'
AND    l.quantity < (
          SELECT 0.2 * AVG(l2.quantity) FROM Lineitem l2
          WHERE l2.partkey = p.partkey);
```

Q18

```
SELECT c.name, c.custkey, o.orderkey, o.orderdate,
       o.totalprice, SUM(l.quantity) AS query18
FROM   Customer c, Orders o, Lineitem l
WHERE  o.orderkey IN
       ( SELECT l3.orderkey FROM (
            SELECT l2.orderkey, SUM(l2.quantity) AS QTY
            FROM Lineitem l2 GROUP BY l2.orderkey ) l3
         WHERE QTY > 100)
AND    c.custkey = o.custkey
AND    o.orderkey = l.orderkey
GROUP BY c.name, c.custkey, o.orderkey, o.orderdate,
        o.totalprice;
```

Q20

```
SELECT s.name, s.address
FROM   Supplier s, Nation n
WHERE  s.suppkey IN
       ( SELECT ps.suppkey
         FROM   Partsupp ps
         WHERE ps.partkey IN
                ( SELECT p.partkey
                  FROM   Part p
                  WHERE  p.name like 'forest%' )
         AND   ps.availqty >
                ( SELECT 0.5 * SUM(l.quantity)
                  FROM   Lineitem l
                  WHERE  l.partkey = ps.partkey
                  AND    l.suppkey = ps.suppkey
                  AND    l.shipdate >= DATE('1994-01-01')
                  AND    l.shipdate < DATE('1995-01-01') ))
AND    s.nationkey = n.nationkey
AND    n.name = 'CANADA';
```

Q19

```
SELECT SUM(l.extendedprice * (1 - l.discount) ) AS revenue
FROM Lineitem l, Part p
WHERE
    (
        p.partkey = l.partkey
        AND p.brand = 'Brand#12'
        AND ( p.container IN ( 'SM CASE', 'SM BOX',
                               'SM PACK', 'SM PKG') )
        AND l.quantity >= 1 AND l.quantity <= 1 + 10
        AND ( p.size BETWEEN 1 AND 5 )
        AND (l.shipmode IN ('AIR', 'AIR REG') )
        AND l.shipinstruct = 'DELIVER IN PERSON'
    )
    OR
    (
        p.partkey = l.partkey
        AND p.brand = 'Brand#23'
        AND ( p.container IN ('MED BAG', 'MED BOX',
                              'MED PKG', 'MED PACK') )
        AND l.quantity >= 10 AND l.quantity <= 10 + 10
        AND ( p.size BETWEEN 1 AND 10 )
        AND ( l.shipmode IN ('AIR', 'AIR REG') )
        AND l.shipinstruct = 'DELIVER IN PERSON'
    )
    OR
    (
        p.partkey = l.partkey
        AND p.brand = 'Brand#34'
        AND ( p.container IN ( 'LG CASE', 'LG BOX',
                               'LG PACK', 'LG PKG') )
        AND l.quantity >= 20 AND l.quantity <= 20 + 10
        AND ( p.size BETWEEN 1 AND 15 )
        AND ( l.shipmode IN ('AIR', 'AIR REG') )
        AND l.shipinstruct = 'DELIVER IN PERSON'
    );
```

Q21

```
SELECT s.name, COUNT(*) AS numwait
FROM   Supplier s, Lineitem l1, Orders o, Nation n
WHERE  s.suppkey = l1.suppkey
AND    o.orderkey = l1.orderkey
AND    o.orderstatus = 'F'
AND    l1.receiptdate > l1.commitdate
AND    (EXISTS (SELECT * FROM Lineitem l2
                WHERE l2.orderkey = l1.orderkey
                AND   l2.suppkey <> l1.suppkey))
AND    (NOT EXISTS
               (SELECT * FROM Lineitem l3
                WHERE  l3.orderkey = l1.orderkey
                AND    l3.suppkey <> l1.suppkey
                AND    l3.receiptdate > l3.commitdate))
AND    s.nationkey = n.nationkey
AND    n.name = 'SAUDI ARABIA'
GROUP BY s.name;
```

Q22

```
SELECT  cntrycode,
        COUNT(*) AS numcust,
        SUM(custsale.acctbal) AS totalacctbal
FROM (
  SELECT SUBSTRING(c.phone, 0, 2) AS cntrycode,
         c.acctbal
  FROM   Customer c
  WHERE  (SUBSTRING(c.phone, 0, 2) IN
              ('13', '31', '23', '29', '30', '18', '17'))
  AND    c.acctbal > (
            SELECT AVG(c2.acctbal)
            FROM   Customer c2
            WHERE  c2.acctbal > 0.00
            AND    (SUBSTRING(c2.phone, 0, 2) IN
                       ('13', '31', '23', '29',
                              '30', '18', '17')))
  AND    (NOT EXISTS (SELECT * FROM Orders o
                      WHERE o.custkey = c.custkey))
  ) custsale
GROUP BY cntrycode;
```

A.1.2 Financial workload

AXF
```
SELECT   b.broker_id, SUM(a.volume-b.volume)
FROM     Bids b, Asks a
WHERE    b.broker_id = a.broker_id
AND      (a.price-b.price > 1000 OR b.price-a.price > 1000)
GROUP BY b.broker_id;
```

BSP
```
SELECT x.broker_id, SUM(x.volume*x.price - y.volume*y.price)
FROM   Bids x, Bids y
WHERE  x.broker_id=y.broker_id AND x.t>y.t
GROUP BY x.broker_id;
```

BSV
```
SELECT x.broker_id, SUM(x.volume*x.price*y.volume*y.price*0.5)
FROM   Bids x, Bids y
WHERE  x.broker_id = y.broker_id
GROUP BY x.broker_id;
```

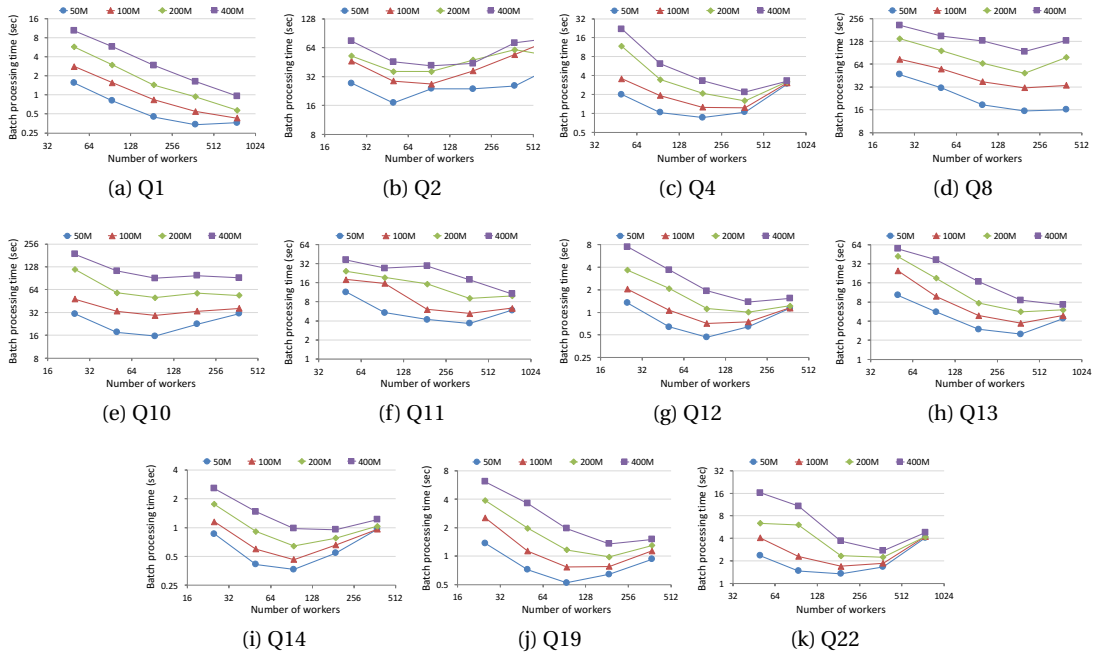
MST
```
SELECT b.broker_id, SUM(a.price*a.volume - b.price*b.volume)
FROM   Bids b, Asks a
WHERE 0.25*(SELECT SUM(a1.volume) FROM Asks a1) >
  (SELECT SUM(a2.volume) FROM Asks a2 WHERE a2.price>a.price)
AND   0.25*(SELECT SUM(b1.volume) FROM Bids b1) >
  (SELECT SUM(b2.volume) FROM Bids b2 WHERE b2.price>b.price)
GROUP BY b.broker_id;
```

PSP
```
SELECT SUM(a.price - b.price)
FROM   Bids b, Asks a
WHERE  b.volume>0.0001*(SELECT SUM(b1.volume) FROM Bids b1)
AND    a.volume>0.0001*(SELECT SUM(a1.volume) FROM Asks a1);
```

VWAP
```
SELECT SUM(b1.price * b1.volume)
FROM   Bids b1
WHERE  0.25 * (SELECT SUM(b3.volume) FROM Bids b3) >
       (SELECT SUM(b2.volume) FROM Bids b2
        WHERE b2.price>b1.price);
```

A.2 Distributed Experiments with Batch Updates (cont.)

Figure A.1 presents more scalability results of the distributed incremental view maintenance of the TPC-H queries.



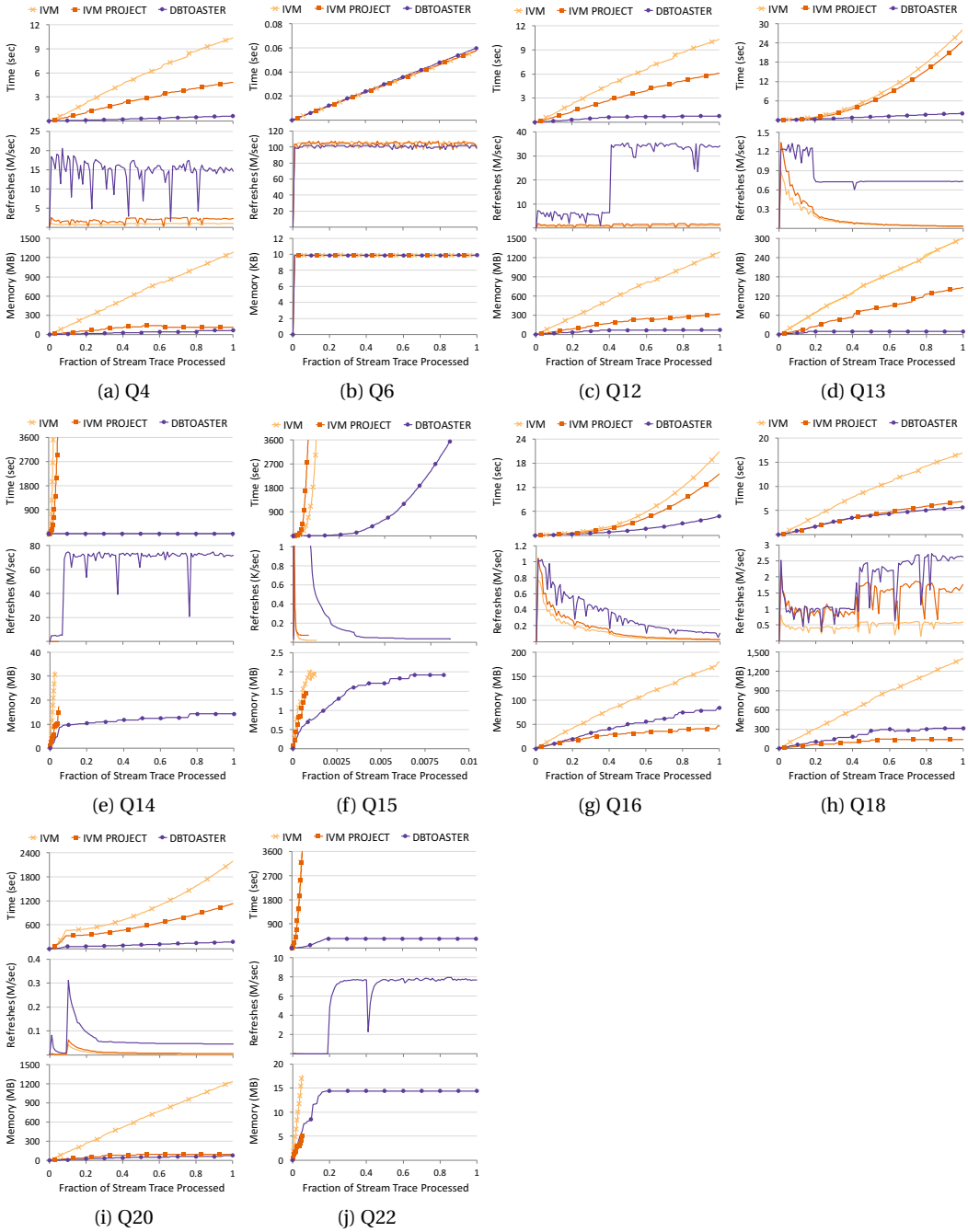
(a) Q1 (b) Q2 (c) Q4 (d) Q8

(e) Q10 (f) Q11 (g) Q12 (h) Q13

(i) Q14 (j) Q19 (k) Q22

Figure A.1 – Strong scalability of the incremental view maintenance of TPC-H queries for different batch sizes (in million of tuples).

A.3 Local Experiments with Single-tuple Updates (cont.)



(a) Q4 (b) Q6 (c) Q12 (d) Q13

(e) Q14 (f) Q15 (g) Q16 (h) Q18

(i) Q20 (j) Q22

Figure A.2 – (a) Join-free query with an equality/inequality correlated nested aggregate in the EXISTS clause. (b) Join-free query. (c) 2-way join. (d) 2-way join with an aggregate query in the FROM clause. (e) 2-way join. (f) 2-way join with an aggregate query in the FROM clause and an inequality-correlated nested aggregate in the EXIST clause. (g) 2-way join with a nested inequality correlated query. (h) 3-way join with an equality correlated nested query and an uncorrelated aggregate in the FROM clause. (i) 2-way join with 3 equality-correlated nested queries. (j) Join-free query with an uncorrelated nested aggregate and a correlated nested aggregate in the EXISTS clause.

A.4 Comparison DBToaster vs. PostgreSQL

	Re-evaluation (PostgreSQL)						IVM (PostgreSQL)						Recursive IVM (DBToaster, C++)						
Batch size	1	10	100	1000	10000	100000	1	10	100	1000	10000	100000	Single	1	10	100	1000	10000	100000
TPC-H 1	10	32	103	187	1003	2889	57	332	2478	9686	12250	12194	1267132	307715	1186926	2812549	4179921	4163385	4372480
TPC-H 2	11	146	427	493	5685	13420	28	125	193	4726	1995	12044	756611	456559	911141	1005175	971154	986810	691260
TPC-H 3	34	111	301	522	2120	5722	17	78	222	61	9100	5676	3736860	1251162	3004637	4401315	5323732	5182684	4580003
TPC-H 4	23	122	400	400	3713	7333	55	93	5857	10223	12550	12971	10076062	1603637	4652895	9012747	10687864	10790913	9752380
TPC-H 5	30	88	253	460	2304	5424	11	174	9	11	153	8665	584261	387568	625805	690475	658595	618632	509490
TPC-H 6	27	88	282	319	2500	6528	58	579	4065	11091	12742	12556	138216710	17017320	44270149	78310773	98176845	116931875	101327791
TPC-H 7	34	117	311	317	2985	7083	7	30	2	16	6908	5944	650650	397643	689015	753051	775288	770259	646018
TPC-H 8	35	114	338	335	2629	5886	28	8	21	19	222	5571	91221	73786	219613	279041	273387	277808	221020
TPC-H 9	30	151	335	513	1952	5999	14	25	22	21	197	1545	104370	78460	90949	109515	102940	86251	76296
TPC-H 10	32	88	322	207	2282	5701	33	165	372	123	7836	12306	2889537	1391237	3605282	5771226	6354245	7050705	5964290
TPC-H 11	22	61	191	366	1930	8734	23	71	220	367	2062	9414	768	776	1923	13500	109603	407547	591716
TPC-H 12	33	109	356	646	3279	4823	49	1050	6248	10837	630	5306	8675929	1678780	3905117	7131341	8236605	7706686	7469474
TPC-H 13	21	74	170	489	345	2578	33	94	283	267	2108	7334	779515	444588	685871	758725	701083	679684	474765
TPC-H 14	26	74	230	313	2322	6556	35	40	56	148	11686	11611	33041606	2769955	13146565	27984674	43468480	51827803	53436252
TPC-H 15	18	74	203	154	1613	4674	N/A	34	38	86	923	3944	17	17	27	52	109	285	964
TPC-H 16	19	58	203	330	1161	2822	18	35	302	1317	5304	10095	123936	115749	131902	121464	108208	75015	58721
TPC-H 17	27	57	88	194	481	666	36	111	564	948	589	1288	379303	210671	256882	208937	279930	155138	131964
TPC-H 18	20	66	207	223	1190	2114	20	18	17	77	676	5881	1133647	572132	1040612	1278945	1272541	1274853	971313
TPC-H 19	31	72	234	329	2218	6139	42	422	502	144	291	11944	1946309	2461229	5829592	8753856	9988084	9737049	8776165
TPC-H 20	6	9	15	18	36	56	11	7	5	21	25	43	977	950	1504	129092	874469	2191422	1871407
TPC-H 21	32	110	347	417	794	7333	31	54	24	10	10583	6797	836800	282532	449838	508128	501657	478923	407540
TPC-H 22	14	45	141	247	1551	5462	12	36	97	298	1252	741	189	183	336	5918	54459	434245	815903
TPC-DS 3	35	92	328	2045	3136	6694	164	1785	4457	5444	6866	8416	12309621	2072493	5945894	8116538	8718851	7740490	6442932
TPC-DS 7	1	14	109	219	1635	5000	98	1104	4262	4438	6721	6694	858649	361726	1024456	1227564	1076421	963048	392581
TPC-DS 19	1	106	132	264	2480	5056	0	78	232	82	7231	6139	30	29	29	29	29	25	7
TPC-DS 27	107	837	3706	5760	7307	8037	76	867	4326	6331	6846	8212	588394	196141	560333	669487	529235	497035	202849
TPC-DS 34	0	4	42	208	1821	4730	1	4	41	173	7510	6350	2803540	1019381	4742812	9219979	10612493	11445739	10689928
TPC-DS 42	9	79	405	196	4322	5594	153	1234	4607	5124	6766	7694	21127917	2589874	7953492	10242024	11325623	9460881	8530989
TPC-DS 43	0	4	42	201	1643	4411	91	710	3232	4702	7246	6972	3103901	589052	3488082	9893883	12851893	14494762	14605017
TPC-DS 46	0	4	43	204	2050	5040	0	1	14	56	6599	6611	185	149	149	149	149	147	90
TPC-DS 52	8	79	406	191	4289	6583	136	869	4266	5066	6808	8121	21005081	2456902	8102885	10552388	11401574	10311449	9074722
TPC-DS 55	31	97	294	875	2444	5444	52	1128	4639	5726	6528	6825	34822881	2343359	7680837	10282103	10993911	10028796	8951066
TPC-DS 68	0	4	42	202	2226	5238	0	1	14	57	6628	6361	182	149	149	149	149	147	88
TPC-DS 73	107	810	3866	5892	7308	8111	139	896	4352	6396	6919	8184	2104283	1028758	4739743	8906107	10620186	10460940	9176666
TPC-DS 79	0	4	39	197	1819	4066	1	4	41	170	5036	6111	838490	427822	1903781	3289301	3869664	3254954	2834858

Table A.1 – Throughput comparison of re-evaluation and incremental maintenance in PostgreSQL and recursive incremental maintenance in generated C++ code for different batch sizes (in tuples per second).

Bibliography

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Apache Mahout. http://mahout.apache.org/.

[3] CERN - Large Hadron Collider. http://home.cern/about/computing. Retrieved 06/06/2016.

[4] Create Indexed Views. http://msdn.microsoft.com/en-us/library/ms191432.aspx.

[5] DBToaster Release 2.2 revision 3387, Nov. 27, 2015. http://www.dbtoaster.org/index.php?page=download.

[6] Expression Trees. https://msdn.microsoft.com/en-us/library/bb397951.aspx. Retrieved 05/29/2016.

[7] Intel Math Kernel Library. https://software.intel.com/en-us/intel-mkl. Retrieved 06/06/2016.

[8] LabVIEW homepage. http://www.ni.com/labview.

[9] Materialized View Concepts and Architecture. http://docs.oracle.com/cd/B28359_01/server.111/b28326/repmview.htm.

[10] MATLAB Signal Processing Toolbox homepage. http://www.mathworks.com/products/signal.

[11] Scaling the Facebook Data Warehouse to 300PB. https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/. Retrieved 06/06/2016.

[12] SQL Server R Services. https://msdn.microsoft.com/en-us/library/mt604845.aspx. Retrieved 05/29/2016.

[13] StreamBase homepage. http://www.streambase.com.

[14] The LINQ Project. http://tinyurl.com/42egdn. Retrieved 05/29/2016.

[15] The r_exec plugin for running R code within SciDB queries. https://github.com/Paradigm4/r_exec. Retrieved 05/29/2016.

[16] TPC-DS Benchmark Specification. Transaction Processing Performance Council, http://www.tpc.org/tpcds/default.asp, 2015.

[17] TPC-H Benchmark Specification. Transaction Processing Performance Council, http://www.tpc.org/tpch/, 2014.

[18] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.

[19] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An Experimental Analysis of Self-adjusting Computation. *TOPLAS*, 32(1), 2009.

[20] Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *SIGMOD*, pages 137–148, 1996.

[21] Charu C. Aggarwal, editor. *Data Streams - Models and Algorithms*, volume 31. Springer, 2007.

[22] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh, and Tolga Yurek. Efficient View Maintenance at Data Warehouses. In *SIGMOD*, pages 417–427, 1997.

[23] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, pages 496–505, 2000.

[24] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *PVLDB*, 5(10):968–979, 2012.

[25] Yanif Ahmad and Christoph Koch. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *PVLDB*, 2(2):1566–1569, 2009.

[26] Alexander Aiken, Joseph M. Hellerstein, and Jennifer Widom. Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *TODS*, 20(1):3–41, 1995.

[27] Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. *IEEE Trans. Inf. Theory*, 46(2):325–343, 2000.

[28] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11):1033–1044, 2013.

[29] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB*, 8(12):1792–1803, 2015.

[30] Mohamed H. Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer. The Extensibility Framework in Microsoft StreamInsight. In *ICDE*, pages 1242–1253, 2011.

[31] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *POPL*, pages 293–302, 1989.

[32] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. *Book chapter*, 2004.

[33] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[34] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR*, pages 363–374, 2007.

[35] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The Multidimensional Database System RasDaMan. In *SIGMOD*, pages 575–577, 1998.

[36] James O. Berger. *Statistical Decision Theory and Bayesian Analysis*. Springer, 1985.

[37] Pavel Berkhin. Survey: A Survey on PageRank Computing. *Internet Mathematics*, 2(1):73–120, 2005.

[38] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, 1986.

[39] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, pages 479–490, 2006.

[40] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *Mobile Data Management*, pages 3–14, 2001.

[41] Paul G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, pages 963–968, 2010.

[42] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1):285–296, 2010.

[43] O Peter Buneman and Eric K Clemons. Efficiently Monitoring Relational Databases. *TODS*, 4(3):368–382, 1979.

[44] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, et al. S-Store: A Streaming NewSQL System for Big Velocity Applications. *PVLDB*, 7(13):1633–1636, 2014.

[45] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4):401–412, 2014.

[46] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *SIGMOD*, page 668, 2003.

[47] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, pages 34–43, 1998.

[48] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD*, 26(1):65–74, 1997.

[49] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing Queries with Materialized Views. In *ICDE*, pages 190–200, 1995.

[50] Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed Automatic Incrementalization. In *PLDI*, pages 299–310, 2012.

[51] Rada Chirkova and Jun Yang. Materialized Views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[52] Chee-Yee Chong and S. P. Kumar. Sensor Networks: Evolution, Opportunities, and Challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.

[53] Henry Cohn, Robert D. Kleinberg, Balázs Szegedy, and Christopher Umans. Group-theoretic Algorithms for Matrix Multiplication. In *FOCS*, pages 379–388, 2005.

[54] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for Deferred View Maintenance. In *SIGMOD*, pages 469–480, 1996.

[55] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Supporting Multiple View Maintenance Policies. In *SIGMOD*, pages 405–416, 1997.

[56] Graham Cormode and S. Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. *TODS*, 30(1):249–278, 2005.

[57] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, pages 647–651, 2003.

[58] Philippe Cudré-Mauroux, Hideaki Kimura, Kian-Tat Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel L. Wang, Magdalena Balazinska, Jacek Becla, David J. DeWitt, Bobbi Heath, David Maier, Samuel Madden, Jignesh M. Patel, Michael Stonebraker, and Stanley B. Zdonik. A Demonstration of SciDB: A Science-Oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[59] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[60] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

[61] Umeshwar Dayal and Nathan Goodman. Query Optimization for CODASYL Database Systems. In *SIGMOD*, pages 138–150, 1982.

[62] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[63] Linzhong Deng. *Multiple-rank Updates to Matrix Factorizations for Nonlinear Analysis and Circuit Design.* PhD thesis, Stanford University, 2010.

[64] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *TOMS*, 16(1):1–17, 1990.

[65] Jack Dongarra and Piotr Luszczek. ScaLAPACK. In *Encyclopedia of Parallel Computing*, pages 1773–1775. 2011.

[66] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *HPDC*, pages 810–818, 2010.

[67] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed Query Processing in a Relational Data Base System. In *SIGMOD*, pages 169–180, 1978.

[68] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, pages 325–336, 2012.

[69] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: The System S Declarative Stream Processing Engine. In *SIGMOD*, pages 1123–1134, 2008.

[70] Thanaa M. Ghanem, Ahmed K. Elmagarmid, Per-Åke Larson, and Walid G. Aref. Supporting Views in Data Stream Management Systems. *TODS*, 35(1), 2010.

[71] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. The Case for a Signal-Oriented Data Stream Management System. In *CIDR*, pages 397–406, 2007.

[72] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. XStream: A Signal-Oriented Data Stream Management System. In *ICDE*, pages 1180–1189, 2008.

[73] Ananth Grama. *Introduction to Parallel Computing*. Pearson Education, 2003.

[74] Timothy Griffin and Leonid Libkin. Incremental Maintenance of Views with Duplicates. In *SIGMOD*, pages 328–339, 1995.

[75] Ashish Gupta and Inderpal Mumick. *Materialized Views*. MIT Press, 1999.

[76] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[77] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD*, pages 157–166, 1993.

[78] Ashish Kumar Gupta, Alon Y. Halevy, and Dan Suciu. View Selection for Stream Processing. In *WebDB*, pages 83–88, 2002.

[79] Himanshu Gupta and Inderpal Singh Mumick. Selection of Views to Materialize in a Data Warehouse. *IEEE TKDE*, 17(1):24–43, 2005.

[80] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, Mohamed Y. Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, Robert Gwadera, Ihab F. Ilyas, Mirette S. Marzouk, and Xiaopeng Xiong. Nile: A Query Processing Engine for Data Streams. In *ICDE*, page 851, 2004.

[81] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Elsevier, 2011.

[82] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[83] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *CIDR*, pages 68–78, 2007.

[84] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[85] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *PODS*, pages 113–124, 1995.

[86] Christian S. Jensen and Richard T. Snodgrass. Temporal specialization. In *Encyclopedia of Database Systems*, pages 3017–3018. 2009.

[87] Sepandar Kamvar, Taher Haveliwala, and Gene Golub. Adaptive Methods for the Computation of PageRank. *Linear Algebra and its Applications*, 386:51–65, 2004.

[88] Yannis Katsis, Yoav Freund, and Yannis Papakonstantinou. Combining Databases and Signal Processing in Plato. In *CIDR*, 2015.

[89] Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Implementing Incremental View Maintenance in Nested Data Models. In *DBPL*, pages 202–221, 1997.

[90] Oliver Kennedy, Yanif Ahmad, and Christoph Koch. DBToaster: Agile Views for a Dynamic Data Management System. In *CIDR*, pages 284–295, 2011.

[91] Ralph Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley, 1996.

[92] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014.

[93] Christoph Koch. Incremental Query Evaluation in a Ring of Databases. In *PODS*, pages 87–98, 2010.

[94] Christoph Koch. Incremental Query Evaluation in a Ring of Databases. Technical report, EPFL, 2013. Technical Report EPFL-REPORT-183766, https://infoscience.epfl.ch/record/183766.

[95] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *VLDBJ*, 23(2):253–278, 2014.

[96] Christoph Koch, Daniel Lupei, and Val Tannen. Incremental Query Evaluation in a Ring of Databases. In *PODS*, 2016.

[97] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[98] Donald Kossmann. The State of the Art in Distributed Query Processing. *CSUR*, 32(4):422–469, 2000.

[99] Yannis Kotidis and Nick Roussopoulos. A Case for Dynamic View Management. *TODS*, 26(4):388–423, 2001.

[100] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[101] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly Sharing for Streamed Aggregation. In *SIGMOD*, pages 623–634, 2006.

[102] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, pages 239–250, 2015.

[103] Amy Nicole Langville and Carl Dean Meyer. Survey: Deeper Inside PageRank. *Internet Mathematics*, 1(3):335–380, 2003.

[104] Per-Åke Larson and Jingren Zhou. Efficient Maintenance of Materialized Outer-Join Views. In *ICDE*, pages 56–65, 2007.

[105] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Static caching for incremental computation. *TOPLAS*, 20(3):546–585, 1998.

[106] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *TODS*, 30(1):122–173, 2005.

[107] David Maier, Jin Li, Peter A. Tucker, Kristin Tufte, and Vassilis Papadimos. Semantics of Data Streams and Operators. In *ICDT*, pages 37–52, 2005.

[108] Arunprasad P. Marathe and Kenneth Salem. Query Processing Techniques for Arrays. *VLDBJ*, 11(1):68–91, 2002.

[109] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[110] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[111] Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha. REX: Recursive, Delta-Based Data-Centric Computation. *PVLDB*, 5(11):1280–1291, 2012.

[112] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR*, 2003.

[113] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[114] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *SOSP*, pages 439–455, 2013.

[115] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[116] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[117] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed Stream Computing Platform. In *ICDMW*, pages 170–177, 2010.

[118] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case Optimal Join Algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[119] Milos Nikolic, Mohammad Dashti Rahmat Abadi, and Christoph Koch. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *SIGMOD*, 2016.

[120] Milos Nikolic, Mohammed Elseidy, and Christoph Koch. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *SIGMOD*, pages 253–264, 2014.

[121] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110, 2008.

[122] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, pages 567–574, 2001.

[123] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In *VLDB*, pages 802–813, 2002.

[124] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *SIGMOD*, pages 61–72, 2012.

[125] Barak A. Pearlmutter and Jeffrey Mark Siskind. Lazy Multivariate Higher-order Forward-mode AD. In *POPL*, pages 155–160, 2007.

[126] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, pages 251–264, 2010.

[127] William H Press. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[128] Dallan Quass and Jennifer Widom. On-Line Warehouse View Maintenance. In *SIGMOD*, pages 393–404, 1997.

[129] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[130] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[131] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *SIGMOD*, pages 447–458, 1996.

[132] Nick Roussopoulos. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *TODS*, 16(3):535–563, 1991.

[133] Kenneth Salem, Kevin S. Beyer, Roberta Cochrane, and Bruce G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD*, pages 129–140, 2000.

[134] Matthias Seeger. Low rank updates for the cholesky decomposition. Technical report, EPFL, 2004.

[135] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex Query Decorrelation. In *ICDE*, pages 450–458, 1996.

[136] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to Architect a Query Compiler. In *SIGMOD*, 2016.

[137] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[138] Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. One Size Fits All? Part 2: Benchmarking Studies. In *CIDR*, pages 173–184, 2007.

[139] Michael Stonebraker, Jacek Becla, David J. DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B. Zdonik. Requirements for Science Data Bases and SciDB. In *CIDR*, pages 173–184, 2009.

[140] Mark Sullivan and Andrew Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *USENIX*, 1998.

[141] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[142] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB*, pages 309–320, 2003.

[143] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[144] Todd L. Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*, pages 96–106, 2014.

[145] Shivaram Venkataraman, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. Using R for Iterative and Incremental Processing. In *HotCloud*, 2012.

[146] Stratis Viglas and Jeffrey F. Naughton. Rate-based Query Optimization for Streaming Information Sources. In *SIGMOD*, pages 37–48, 2002.

[147] Florian M. Waas. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *BIRTE*, 2008.

[148] R. Clinton Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *PPSC*, 1999.

[149] Janet L. Wiener, Himanshu Gupta, Wilburt Labio, Yue Zhuge, Hector Garcia-Molina, and Jennifer Widom. A System Prototype for Warehouse View Maintenance. In *Workshop on Materialized Views*, pages 26–33, 1996.

[150] Jun Yang and Jennifer Widom. Incremental Computation and Maintenance of Temporal Aggregates. *VLDBJ*, 12(3):262–283, 2003.

[151] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[152] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[153] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, pages 423–438, 2013.

[154] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*, pages 265–276, 2014.

[155] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. iMapReduce: A Distributed Computing Framework for Iterative Computation. *Journal of Grid Computing*, 10(1):47–68, 2012.

[156] Yi Zhang, Herodotos Herodotou, and Jun Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR*, 2009.

[157] Ying Zhang, Martin L. Kersten, and Stefan Manegold. SciQL: Array Data Processing Inside an RDBMS. In *SIGMOD*, pages 1049–1052, 2013.

[158] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. Lazy Maintenance of Materialized Views. In *VLDB*, pages 231–242, 2007.

[159] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient Exploitation of Similar Subexpressions for Query Processing. In *SIGMOD*, pages 533–544, 2007.

[160] Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta Cochrane, Hamid Pirahesh, Latha S. Colby, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *ICAC*, pages 180–188, 2004.

Miloš Nikolić

Chemin des Noutes 17
1023 Crissier
Switzerland

people.epfl.ch/milos.nikolic
milos.nikolic@epfl.ch
+41 (0)76 494.47.83

INTERESTS

I am interested in data management, stream processing, and incremental computation.

My research focuses on the incremental processing of complex analytical queries, such as SQL queries and linear algebra programs, in local and distributed streaming environments using novel approaches to query optimization and compilation.

EDUCATION

École Polytechnique Fédérale de Lausanne (EPFL) 2010-2016 (expected)
Ph.D. in Computer Science
Advisor: Christoph Koch

University of Novi Sad, Faculty of Technical Sciences 2002-2008
M.Sc. in Electrical and Computer Engineering
Ranked 1st in the graduating class (GPA: 10/10)

RESEARCH EXPERIENCE

Graduate Student Researcher September 2010–Present
EPFL, Data Analysis Theory and Applications Lab Lausanne, Switzerland
Advisor: Christoph Koch

Incremental computation of complex analytical queries:

- I worked on designing and building an SQL compiler, called DBToaster, that turns database queries into high-performance stream processing engines by using recursive incremental view maintenance and modern compilation techniques.
- I studied the effects of batch size and code specialization on the incremental processing of SQL queries in local and distributed streaming environments.
- I developed a framework, called Linview, that incrementally executes iterative linear algebra programs by using matrix factorizations to contain maintenance costs.

Data-driven coordination on the social web:

- In collaboration with researchers at Cornell University, I developed a system for data-driven coordination in social applications that uses declarative primitives for expressing users' coordination constraints with transactional semantics.

Intern, Microsoft Research Summer 2015
Database Group Redmond, US
Advisor: Jonathan Goldstein

- I designed and implemented digital signal processing functionality inside a stream processing engine. The developed system unifies relational and signal processing into one data and query model, supports real-time and offline analysis, and offers performance competitive with best signal processing tools and orders of magnitude better than existing data managements systems with signal processing capabilities.

Research Assistant February 2008–August 2010
Department of Applied Computer Science and Informatics Novi Sad, Serbia
Advisor: Miroslav Hajdukovic

- I designed and implemented distributed algorithms for the structural analysis of physical objects using the finite strip method.

PUBLICATIONS

Data Management Systems

- Milos Nikolic, Mohammad Dashti, Christoph Koch, "How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates", *SIGMOD 2016.*
- Milos Nikolic, Mohammed El Seidy, Christoph Koch, "LINVIEW: Incremental View Maintenance for Complex Analytical Queries", SIGMOD 2014.
- Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, Amir Shaikhha, "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views", The VLDB Journal, 23(2):253-278, 2014.
- Yanif Ahmad, Oliver Kennedy, Christoph Koch, Milos Nikolic, "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views", VLDB 2012.

  Invited to the "*Best of VLDB 2012*" issue of the VLDB Journal.
- Nitin Gupta, Milos Nikolic, Sudip Roy, Gabriel Bender, Lucja Kot, Johannes Gehrke, Christoph Koch, "Entangled Transactions", VLDB 2011.

Heterogeneous Parallel Programming

- Milos Nikolic, Miroslav Hajdukovic, Dragan Milasinovic, Danica Goles, Petar Maric, Zarko Zivanov, "Hybrid MPI/OpenMP Cloud Parallelization of Harmonic Coupled Finite Strip Method Applied on Reinforced Concrete Prismatic Shell Structure", Advances in Engineering Software, 84:55-67, 2015.
- Miroslav Hajdukovic, Dragan Milasinovic, Danica Goles, Milos Nikolic, Petar Maric, Zarko Zivanov, Predrag Rakic, "Cloud Computing Based MPI/OpenMP Parallelization of Harmonic Coupled Finite Strip Method Applied on Reinforced Concrete Prismatic Shell Structure", PARENG 2013.
- Milos Nikolic, Dragan Milasinovic, Zarko Zivanov, Petar Maric, Miroslav Hajdukovic, Aleksandar Borkovic, Igor Milakovic, "MPI/OpenMP Parallelisation of the Harmonic Coupled Finite-Strip Method", PARENG 2011.
- Predrag Rakic, Dragan Milasinovic, Zarko Zivanov, Zorica Suvajdzin, Milos Nikolic, Miroslav Hajdukovic, "MPI-CUDA Parallelization of a Finite-Strip Program for Geometric Nonlinear Analysis – A Hybrid Approach", Advances in Engineering Software, 42(5):273-285, 2011.
- Dragan Milasinovic, Zarko Zivanov, Predrag Rakic, Zorica Suvajdzin, Milos Nikolic, Miroslav Hajdukovic, Aleksandar Borkovic, Igor Milakovic, "A Finite-Strip Analysis of Nonlinear Shear-Lag Effect Supported by Automatic Visualization", ECT 2010.

In Preparation

- Milos Nikolic, Badrish Chandramouli, Jonathan Goldstein, "Enabling Signal Processing Over Data Streams".

Theses

- Milos Nikolic, "Dynamic Software Performance Analysis Using DTRACE Tool", M.Sc. Thesis, Faculty of Technical Sciences, University of Novi Sad, 2008.

AWARDS & HONORS

- EPFL IC Teaching Assistant Award, 2014
- "Best of VLDB 2012" invitation to the VLDB Journal
- Best Graduate Student Award at the Faculty of Technical Sciences, 2008
- Scholarship of the Serbian Government, 2002-2008
- Scholarship of the University of Novi Sad, 2004
- 1st Prize at National Competition in Advanced Mathematics, 2004
- 3rd Prize at National Competition in Fundamentals of Electrical Engineering, 2003

TEACHING EXPERIENCE

Teaching Assistant (EPFL)

- Principles of Computer Systems — Fall 2015, 2014, 2013
  Graduate level
  Instructors: Ed Bugnion, Christoph Koch, Bryan Ford (2015), Katerina Argyraki (2014, 2013), George Candea (2013)
- Big Data (Advanced Databases) — Spring 2014, 2013, 2012
  Graduate level
  Instructor: Christoph Koch

  I received the *EPFL Teaching Assistant Award 2014* in appreciation of my work as a teaching assistant for the Big Data course.
- Algorithms — Fall 2012
  2nd year undergraduate level
  Instructors: Omid Etesami and Ola Svensson

Teaching Assistant (University of Novi Sad)

- Operating Systems — Spring 2010, 2009
  2nd and 3rd year undergraduate levels — Fall 2009, 2008
  Instructor: Miroslav Hajdukovic
- Computer Architecture — Spring, 2008
  1st year undergraduate level
  Instructor: Miroslav Hajdukovic

REFERENCES

Available upon request.