Just-in-time Analytics Over Heterogeneous Data and Hardware

THÈSE N^{O} 8077 (2017)

PRÉSENTÉE LE 8 DÉCEMBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES ET APPLICATIONS DE TRAITEMENT DE DONNÉES MASSIVES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Manolis KARPATHIOTAKIS

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury
Prof. A. Ailamaki, directrice de thèse
Dr F. Özcan, rapporteuse
Prof. L. Fegaras, rapporteur
Prof. K. Aberer, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

The struggle itself toward the heights
is enough to fill a man's heart.
One must imagine Sisyphus happy.
— Albert Camus

To Eleni, who had to survive life with a PhD candidate.
To my parents and to my sister.

Acknowledgements

Putting together this thesis required the support and feedback of numerous people; without them, neither my PhD journey nor its destination would have been the same.

I would like to thank my advisor, *Anastasia (Natassa) Ailamaki*, for her guidance and advice. Natassa always looks out for her students. From my very first meeting with her, I realized two undisputed facts: She has an infectious enthusiasm, and she would do anything to ensure that her students can utilize their PhD to pursue a happy life. At the same time, she urges her students to never settle for mediocrity, but instead strive for excellence in all aspects of their research and life. I thus strongly believe that Natassa's guidance has shaped me professionally.

I would also like to thank the members of my thesis committee, both for finding the time to serve in my committee, as well as for their constructive comments throughout my PhD life. Specifically, I would like to thank *Fatma Ozcan*, who acted as my mentor during an internship at IBM and helped shape a significant part of my thesis, *Leonidas Fegaras*, whose own work was highly influential in my thesis, *Karl Aberer*, who has provided feedback to me ever since my first year at EPFL, and *Willy Zwaenepoel*, who accepted to act as the thesis jury president and encouraged me during the thesis exam.

Through the years of my PhD, I have been fortunate to collaborate with a stellar group of people, namely *Miguel Branco*, *Ioannis Alagiannis*, *Matt Olma*, *Stella Giannakopoulou*, *Manos Athanassoulis*, *Raja Appuswamy*, *Danica Porobic*, *Thomas Heinis*, *Benjamin Gaidioz*, *Avrilia Floratou*, and *Periklis Chrysogelos*, and learn new skills from every one of them; I thank you all for our collaboration. Besides the direct collaborators, I have to thank the entire DIAS lab family for making life on the 2nd floor of the EPFL BC building enjoyable: *Adrian, Angelos, Ben, Cesar, Danica, Darius, Diane, Dimitra, Eleni, Erietta, Erika, Farhan, Fotini, George, Ioannis, Iraklis, Lionel, Manos, Matt, Miguel, Mirjana, Odysseas, Panagiotis, Periklis, Pinar, Radu, Raja, Rakesh, Renata, Satya, Snow, Stella, Tahir, Thomas,* and *Utku* were always there to provide feedback / chat / gossip. Likewise, *Amir*, *Mohammed*, and *Yannis* have been great friends during and after our EPFL BC 2nd floor tenure. Ioannis, Manos, and Matt (to his dismay) formed the Greek "frappe coffee / pop culture" group. Matt is the most creative researcher / entrepreneur / handyman / person I know. Iraklis was the Thai cuisine aficionado, and Raja the Justin Bieber one. Mirjana had to listen to my made-up Serbian phrases. Danica and Pinar are my academic big sisters, always looking out for me; I thank you all.

Acknowledgements

Numerous people helped make Lausanne feel like home. Alex, Christos, Stefanos, Natassa, and Ioannis are the loudest of them; Pavlos, Nathalie, and Danae are the kindest; Matt and Stella are the jolliest. Panagiotis and Onur are the human dynamos. Stavros is the go-to guy for bitter beer and for football; Javier the one for random drunken discussions. Katerina, Loukia, Kyveli, Evi, Giannis, Apostolis, George (multiple of you!), Vasilis (also multiple!), Rafa, Farah, Goran, and many more – thank you all for being there.

My parents and my sister have always supported me unconditionally through my N years of studies. My father repeatedly tried to convince me to pursue a PhD, and is the only person more anxious than me during my deadlines; my mother balances him (and me) out; my sister forwards me Internet memes to relieve my stress. Your support has been invaluable.

Last but certainly not least, I'd like to thank my wife Eleni. In the past 5 years, Eleni has had to endure living in a tiny apartment with a workaholic, acting as comic relief to raise my spirits in times of failure, and baking insane amounts of cookies for similar reasons, all while being an infinite source of positive energy. This thesis is for you.

This research has been supported by grants from the School of Computer and Communication Sciences, EPFL, the Swiss National Science Foundation, project No. CRSII2 136318/1, "Trustworthy Cloud Storage", the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa), and an IBM PhD Fellowship Award.

Lausanne, 1 November 2017 M. K.

Abstract

Industry and academia are continuously becoming more data-driven and data-intensive, relying on the analysis of a wide variety of datasets to gain insights. At the same time, data variety increases continuously across multiple axes. First, data comes in multiple formats, such as the binary tabular data of a DBMS, raw textual files, and domain-specific formats. Second, different datasets follow different data models, such as the relational and the hierarchical one. Data location also varies: Some datasets reside in a central "data lake", whereas others lie in remote data sources. In addition, users execute widely different analysis tasks over all these data types. Finally, the process of gathering and integrating diverse datasets introduces several inconsistencies and redundancies in the data, such as duplicate entries for the same real-world concept. In summary, heterogeneity significantly affects the way data analysis is performed.

In this thesis, we aim for *data virtualization*: Abstracting data out of its original form and manipulating it regardless of the way it is stored or structured, without a performance penalty. To achieve data virtualization, we design and implement systems that **i)** mask heterogeneity through the use of heterogeneity-aware, high-level building blocks and **ii)** offer fast responses through on-demand adaptation techniques.

Regarding the high-level building blocks, we use a *query language and algebra* to handle multiple collection types, such as relations and hierarchies, express transformations between these collection types, as well as express complex *data cleaning tasks* over them. In addition, we design a *location-aware compiler and optimizer* that masks away the complexity of accessing multiple remote data sources.

Regarding on-demand adaptation, we present a design to produce a new system *per query*. The design uses *customization mechanisms* that trigger runtime code generation to mimic the system most appropriate to answer a query fast: Query operators are thus created based on the query workload and the underlying data models; the data access layer is created based on the underlying data formats. In addition, we exploit emerging hardware by customizing the system implementation based on the available heterogeneous processors – CPUs and GPGPUs. We thus pair each workload with its ideal processor type. The end result is a *just-in-time database system* that is specific to the query, data, workload, and hardware instance.

This thesis redesigns the data management stack to natively cater for data heterogeneity and exploit hardware heterogeneity. Instead of centralizing all relevant datasets, converting them to a single representation, and loading them in a monolithic, static, suboptimal system, our design embraces heterogeneity. Overall, our design decouples the type of performed analysis from the original data layout; users can perform their analysis across data stores, data models, and data formats, but at the same time experience the performance offered by a custom system that has been built on demand to serve their specific use case.

Keywords: data management, database management systems, data analytics, analytical processing systems, query processing, query compilation, code generation, heterogeneous data, data variety, data virtualization, hybrid transactional/analytical processing, real-time analytics, data lakes, ETL, GPU databases

Résumé

Les secteurs industriel et universitaire continuent de devenir de plus en plus data-driven and data-intensive, dépendant de l'analyse d'une grande variété de données pour faire de nouvelles découvertes. En même temps, la variété des données augmente continuellement, et ce, sur plusieurs axes. Tout d'abord, les données existent en plusieurs formats, comme les SGBD, dont les données sont enregistrée sous forme de tables binaires, les fichiers textes purs et les formats spécifiques au domaine. En second lieu, différents set de données suivent différents modèles de données, par example relationnels ou hiérarchiques. L'emplacement des données varie également : certains datasets résident dans un "lac de données" central, tandis que d'autres résident dans des sources de données distantes. En plus, les utilisateurs exécutent des tâches d'analyse largement différentes sur tous ces types de données. Dernièrement, la procédure pour collecter et intégrer ces sets de données variés introduit plusieurs inexactitudes dans les données, telles que les entrées en double pour le même concept réel. En résumé, l'hétérogénéité affecte de manière significative la façon dont l'analyse des données est effectuée.

Le but de cette thèse et la *virtualisation des données* : Abstraire les données de leur forme originale et les manipuler sans affecter les performances. Pour réaliser la virtualisation des données, nous concevons et mettons en œuvre des systèmes qui **i)** masquent l'hétérogénéité en utilisant des blocs de construction de haut niveau, qui prennent en compte cette hétérogénéité, et **ii)** qui offrent des réponses rapides grâce aux techniques d'adaptation à la demande.

En ce qui concerne les blocs de construction de haut niveau, nous utilisons un *langage de requête et une algèbre* pour gérer plusieurs types de collection, comme les relations et les hiérarchies, pour exprimer des transformations entre ces types de collection, ainsi que pour exprimer des tâches de *nettoyage de données* complexes sur eux. En plus, nous concevons un *compilateur sensible au placement et un optimisateur* qui améliore la complexité de l'accès à plusieurs sources de données distantes.

En ce qui concerne l'adaptation à la demande, nous présentons un design pour produire un nouveau système *par requête*. La conception utilise des *mécanismes d'adaptation* qui déclenchent la génération de code d'exécution pour imiter le système le plus approprié pour répondre rapidement à une requête : Les opérateurs de requête sont ainsi créés en fonction de la charge de travail de la requête et des modèles de données sous-jacents ; la couche d'accès

aux données est créée en fonction des formats de données sous-jacents. En plus, nous exploitons les matériels émergents en personnalisant l'implémentation du système en fonction des processeurs hétérogènes disponibles - CPU et GPGPUs. Nous combinons donc chaque charge de travail avec son type de processeur idéal. Le résultat final est un *système de base de données just-in-time*, spécifique à la requête, aux données, à la charge de travail et au matériel disponible.

Cette thèse redéfini la pile de gestion des données pour couvrir nativement l'hétérogénéité des données et exploiter l'hétérogénéité du matériel informatique. Plutôt que de centraliser tous les sets de données, les transformer en une seule représentation, et les charger dans un système monolithique, statique, sous-optimal, notre design embrasse l'hétérogénéité. En general, notre design découple le type d'analyse effectué de la disposition de données originale ; les utilisateurs peuvent effectuer leur analyse dans des data stores, des modèles de données, et des formats de données différents, tout en profitant des performances offertes par un système qui a été construit à la demande pour servir leur cas d'utilisation spécifique.

Mots clefs : gestion de données, systèmes de gestion de base de données, analyse de données, système de traitement de requêtes analytiques, traitement de requêtes, compilation de requêtes, génération de code, données hétérogènes, variété de données, virtualisation de données, traitement transactionnel / analytique hybride, analyse en temps réel, lacs de données, ETL, bases de données GPU

Contents

Acknowledgements i

Abstract (English/Français) iii

List of figures xi

List of tables xv

1 Introduction 1
- 1.1 Motivating Applications . . . 2
- 1.2 Pitfalls of heterogeneity . . . 2
- 1.3 Thesis Statement and Contributions . . . 4
  - 1.3.1 The end goal: Data virtualization . . . 4
  - 1.3.2 Thesis Roadmap . . . 6

2 Background 7
- 2.1 Raw Data Analytics . . . 7
  - 2.1.1 Traditional: Loading and Accessing Data . . . 7
  - 2.1.2 Accessing Data through External Tables . . . 8
  - 2.1.3 Querying Data *In Situ* . . . 8
- 2.2 Query Languages for Diverse Data Models . . . 9
- 2.3 Analytical Query Processing . . . 13
- 2.4 Query Processing on Emerging Server Hardware . . . 14
  - 2.4.1 Generalization of GPGPUs . . . 14
  - 2.4.2 Specialization of CPUs . . . 16
- 2.5 Data Cleaning . . . 17
  - 2.5.1 Data Cleaning Operations . . . 17
  - 2.5.2 Data Cleaning Systems & Techniques . . . 18

3 Just-in-time Access Paths 21
- 3.1 Introduction . . . 21
- 3.2 Preliminaries: Accessing Data through Positional Maps . . . 24
- 3.3 The RAW Query Engine . . . 25
- 3.4 Adapting to raw data . . . 27
  - 3.4.1 Just-In-Time Access Paths . . . 27

3.4.2 Evaluating raw data access strategies . . . 30
3.5 When To Load Data . . . 33
3.5.1 Shredding Columns . . . 34
3.5.2 Full Columns vs. Column Shreds . . . 35
3.5.3 Column Shreds Tradeoffs . . . 37
3.6 Use Case: The Higgs Boson . . . 41
3.7 Summary . . . 45

4 Just-in-time Query Engines **47**
4.1 Introduction . . . 47
4.2 Related Work . . . 49
4.3 An expressive query algebra . . . 51
4.4 The Architecture of Proteus . . . 53
4.5 On-demand query engines . . . 54
4.5.1 An Engine per Query . . . 55
4.5.2 A Custom Data Access Layer per Query . . . 57
4.6 Adapting storage to workload . . . 62
4.7 Experimental Evaluation . . . 64
4.7.1 Specializing the Query Engine on Demand . . . 65
4.7.2 Adapting to a Real-world Workload . . . 71
4.8 Summary . . . 74

5 Big Data Virtualization **77**
5.1 Introduction . . . 77
5.2 Related Work . . . 80
5.3 Motivation and Background . . . 81
5.4 System-PV . . . 84
5.5 Compiling Cross-store Queries . . . 85
5.5.1 Exposing a Virtual Schema . . . 86
5.5.2 Querying over a Virtual Schema . . . 88
5.6 A Two-phase Optimizer for Cross-store Analytics . . . 89
5.6.1 Phase I: SQL Optimization . . . 90
5.6.2 Phase II: Source-aware Optimization . . . 91
5.7 Experimental Evaluation . . . 94
5.7.1 Experimental Setup . . . 95
5.7.2 System-PV vs. Spark . . . 96
5.7.3 System-PV Performance . . . 98
5.8 Perspectives . . . 99
5.8.1 System-PV for enterprise workloads . . . 100
5.8.2 Optimizing SQL-on-Hadoop performance over multiple sources . . . 100
5.9 Summary . . . 101

6 Unified Scale-Out Data Cleaning **103**
6.1 Introduction . . . 103
6.2 A unified representation . . . 105
6.2.1 Data cleaning operations . . . 106
6.2.2 From data cleaning operations to code . . . 106
6.3 Cleaning data using monoids . . . 107
6.3.1 Optimizations at the monoid level . . . 107
6.3.2 Expressive Power: Mapping cleaning building blocks to the monoid calculus . . . 109
6.3.3 The CleanM language . . . 111
6.4 Unified algebraic optimization . . . 114
6.5 Executing data cleaning tasks . . . 115
6.6 CleanDB: A data cleaning system . . . 117
6.7 Experimental Evaluation . . . 118
6.7.1 Optimizations at the monoid level . . . 119
6.7.2 Optimizations at the algebra level . . . 121
6.7.3 Optimizations at the physical level . . . 123
6.8 Summary . . . 127

7 Looking forward: HTAP on Heterogeneous Hardware **129**
7.1 Introduction . . . 129
7.2 Database engines on emerging hardware . . . 130
7.3 The case for H^2TAP . . . 131
7.4 CALDERA: An H^2TAP query engine . . . 134
7.5 Evaluation . . . 138
7.5.1 HTAP with software snapshotting . . . 138
7.5.2 OLTP with message passing . . . 140
7.5.3 Data sharing with PAX . . . 141
7.6 Summary . . . 143

8 The Big Picture **145**
8.1 Unconditional data virtualization: What we did . . . 146
8.2 Unconditional data virtualization: Next steps . . . 146

A **151**
A.1 Spam Analysis Queries . . . 152

Bibliography **175**

Curriculum Vitae **177**

List of Figures

1.1 The data analysis stack, and the heterogeneity-related challenge in each layer. . 3

2.1 Scan execution time under Fermi/Maxwell GPUs. 15

3.1 JIT access paths vs. In Situ and DBMS approaches: Cold and warm run of a query over CSV data. 32

3.2 JIT access paths vs. In Situ and DBMS approaches: For binary files, JIT access paths outperform traditional in situ query processing. 33

3.3 Benefits from adapting to data: Unrolling the main loop, simplifying parsing and data type conversions reduce the time spent "preparing" raw data. 33

3.4 "Full columns" vs. "Column Shreds". "Full columns": all column values are pre-loaded into columnar structures. "Column shreds": column pieces are only built as needed: in the example, Col2 is only loaded with the rows that passed the filter condition on Col1. 34

3.5 "Full columns" vs. "Column Shreds" - CSV. For the 2nd query over a CSV file, column shreds are always faster or exactly the same as full columns, as only elements of column 11 that pass the predicate are loaded from the file. 35

3.6 "Full columns" vs. "Column Shreds" - Binary. For the 2nd query over a binary file, we see the same behavior as for CSV: use of column shreds is always faster than use of full columns or exactly the same for 100% selectivity. 35

3.7 "Full columns" vs. "Column Shreds". CSV files with floating-point numbers carry a higher data type conversion cost. The DBMS case is significantly faster. 37

3.8 "Full columns" vs. "Column Shreds". The binary format requires no conversions, so the absolute difference between DBMS and column shreds is very small. . . 37

3.9 Shredding Policies: Creating shreds of requested nearby columns in one step is beneficial when accessing raw data in multiple steps is costly. 38

3.10 Shredding Policies: Possible points of column population based on join side. . 39

3.11 Shredding Policies: If the column to be projected is on the "pipelined" side of the join, then delaying its creation is a better option. 40

3.12 Shredding Policies: If the projected column is on the "breaking" side, picking its point of creation depends on the join selectivity. 40

3.13 Simplified version of the ROOT query plan. The overall query is depicted in steps. 43

3.14 Data representation in ROOT and RAW. The representation that RAW uses allows vectorized processing. . . . 44

4.1 Query involving unnest operators: Without them, the operators higher in the tree would have to process BLOBs repeatedly every time they need a nested value. 52
4.2 The architecture of Proteus. . . . 53
4.3 Example of a query plan and of the generated (pseudo-) code. Once the scan operator places needed fields in virtual buffers, they are used to evaluate the filtering expression. . . . 57
4.4 Example of a structural index for a JSON object. . . . 59
4.5 Projection-intensive queries over JSON data. . . . 66
4.6 Projection-intensive queries over binary relational data. . . . 66
4.7 Selection queries over JSON data. . . . 67
4.8 Selection queries over binary relational data. . . . 67
4.9 Join and unnest queries over JSON data. . . . 69
4.10 Join and unnest queries over binary relational data. . . . 69
4.11 Aggregate queries over JSON data. . . . 70
4.12 Aggregate queries over binary relational data. . . . 70
4.13 Effect of caching on i) a projection query and on b) a selection query over JSON data. . . . 71
4.14 For a spam analysis workload, Proteus outperforms the other systems in the majority of queries due to i) its lightweight, specialized-on-demand code paths, and ii) the caches it builds as a side-effect of query execution. . . . 72

5.1 Typical scenario in a data lake: Analyzing recent, actively updated data along with historical data. . . . 82
5.2 Architecture of (a) Spark SQL and of (b) System-PV. Dotted boxes in (b) represent extensions. . . . 84
5.3 System-PV Pipeline. . . . 88
5.4 Virtual plan of our running example (a), and its corresponding grounded plan (b). 89
5.5 Query plan simplification during source-aware optimization. . . . 92
5.6 Join pushdown rewriting during source-aware optimization. . . . 93
5.7 Range query rewriting during source-aware optimization: Data accesses become parallelizable. . . . 94
5.8 Query Plan Quality: The SQL Optimizer of System-PV picks the best candidate plan (No. 1), whereas Spark's Catalyst optimizer picks plan No. 9. . . . 96
5.9 Spark vs. System-PV: Spark is unable to keep up with System-PV even for very selective queries. . . . 97
5.10 System-PV performance for various data placement configurations and query selectivities. . . . 98

6.1 Algebraic plans for our running example, and optimized rewritten plans that coalesce operators and share work. . . . 115

6.2 The architecture of CleanDB. 117
6.3 Different configurations of CleanDB for term validation. 120
6.4 Accuracy of term validation as the noise increases. 120
6.5 Unified data cleaning: CleanDB rewrites three cleaning operations into a single one, and avoids duplicate work. 122
6.6 Cost of checking for violations of functional dependencies over TPC-H. 124
6.7 Cost of checking for violations of functional dependencies over Tax. 125
6.8 Duplicate elimination over simplified representations of DBLP: Spark SQL was unable to terminate when cleaning the original dataset. 126
6.9 Duplicate elimination over Customer and MAG. 126

7.1 H^2TAP deployed over emerging server hardware. 132
7.2 The hierarchical data organization of Caldera for a columnar data layout, and the in-memory state after a transaction has updated table T^a. Superscripts represent epochs. 135
7.3 GPU-powered Caldera vs. CPU-powered columnar engines for Q6 of TPC-H. Time for Caldera includes data transfer costs. 138
7.4 OLTP transaction throughput in the presence of OLAP queries as we vary the OLTP working set and the degree of data freshness. 138
7.5 Execution time of OLAP queries in the presence of OLTP queries. All OLAP queries share a single snapshot, but OLTP-triggered copy-on-write stresses memory bandwidth. 139
7.6 Execution time of OLAP queries and throughput of OLTP queries. We increase the number of queries that share a snapshot from 10 to 100. Increasing snapshot sharing improves performance. 139
7.7 TPC-C scalability as the number of cores increase. 140
7.8 Throughput as the percentage of multi-site transactions increases. 140
7.9 Comparing the efficiency of different data layouts for GPU-based computations. 142
7.10 Comparing different data layouts when all data is GPU resident. 142

List of Tables

2.1 The monoid comprehension calculus 11
2.2 The operators of the nested relational algebra. 12
2.3 Processing power, memory capacity, and interconnection bandwidth of consumer-grade NVIDIA graphics cards across generations 15

3.1 Hardware setup for experiments evaluating RAW. 30
3.2 Execution time of the 1st query over a table with 120 columns of integers and floating-point numbers. A traditional DBMS is significantly slower in the 1st query due to data loading. 36
3.3 Comparison of hand-written C++ Higgs Analysis with the RAW version. 44

4.1 The input plug-in API of Proteus. 58
4.2 Execution time per Symantec workload phase. 74

5.1 Operators used in the view definitions of System-PV. 86

6.1 Translation of algebraic operators to Spark operators. Bold parts introduce new Spark operators or deviate from the translation that Spark SQL would have performed. 116
6.2 Accuracy of term validation approaches over the DBLP dataset. 121
6.3 Overhead introduced by performing syntactic transformations in a plain query. The optimizer of CleanDB applies both operations in one go and reduces overhead by ~ 2×. 123
6.4 Denial constraints involving inequalities as the dataset size increases. All systems beside CleanDB fail to terminate. 124

1 Introduction

Whether in business or in science, the driver of many big data applications is the need for analyzing vast amounts of heterogeneous data to develop new insights. Examples include analyzing medical data to improve diagnosis and treatment, scrutinizing workflow data to understand and optimize business processes, analyzing stock market tickers to support financial trading, etc. Yet, as different as these examples are, their core challenges revolve around providing unified access to data from heterogeneous sources, which remains a formidable challenge today [34, 65, 99, 118, 129, 245] because the datasets to be analyzed typically come in a variety of formats and models, and can reside in a variety of locations / data stores.

State-of-the-art approaches for data analysis have relied on placing all data, originally stored in heterogeneous file formats located at different sources, in one data warehouse. In this process, semantic integration approaches [97] help to map semantically equivalent data from different data sources on a common schema. Physical integration, on the other hand, is commonly addressed by first transforming all heterogeneous data into a common format and then copying and integrating it into a data warehouse. Transforming and integrating all data into a warehouse, however, is no longer an option for a growing number of applications. For example, in many scenarios, institutions owning the data want to retain full control of data, for legal or ethical reasons. In addition, transforming and loading the data into a warehouse is a considerable time investment that is unlikely to pay off as not all data may be accessed, while it bears the risk of vendor lock-in; migrating datasets from a proprietary system to another entails a substantial switching cost in terms of human and computational resources [191]. Furthermore, statically transforming all data into one common format and relying on a single, general-purpose query engine impedes query execution, because different query classes benefit from class-specific i) execution engines and ii) data layouts for efficient query processing. Finally, for applications that opt to operate over a single, aggregating data warehouse, preserving data freshness requires a continuous synchronization effort to propagate updates on the original data sources to the data warehouse in a timely manner; applications thus often ignore the fresh tail end of data that resides in the original sources and operate over stale data.

1.1 Motivating Applications

One of the key visions of the Human Brain project (HBP [175]) is to improve diagnosis and treatment of brain related diseases. Defining sound disease characterizations of brain diseases shared by patients is a necessary first step that requires a representative and large enough sample of patient data. Researchers in the HBP consequently must access data from multiple hospitals in order to perform their analysis over a large sample. Enabling access to heterogeneous data at different hospitals, however, is a massive integration challenge.

Integrating all patient data into one warehouse, i.e., transforming it physically and semantically into the same representation and moving it into one administrative location, seems to be the most straightforward approach to enable data analysis. Nevertheless, patient data appears in multiple, heterogeneous data formats; results from various instruments or processing pipelines are stored as JSON, CSV, medical image formats containing arrays, etc. Also, frequent updates occur over medical records. In addition, practitioners launch different types of analysis over the same data. Thus, importing all data into a warehouse is impractical, as many database researchers have recognized [29, 36, 78, 135, 141, 144, 145]. Instead, data remain in their original sources, regardless of whether these sources are loaded databases or raw files. In summary, the major data management challenge lies in optimizing the physical integration of data stored in heterogeneous formats (e.g., database tables, CSV, JSON, etc.) to efficiently support heterogeneous queries.

Similar challenges are common in other applications as well. Banks, for example, operate large numbers of databases and data processing frameworks. The banking sector thus requires a single data access layer that different functional domains (e.g., Trading, Risk, Settlement) can manage, but no such data access layer is available. Existing data processing systems are impractical to use across such a heterogeneous, complex data ecosystem. Furthermore, regulations require banks to keep raw data and correlate it directly with the trade life cycle. Accessing all data in its original form, on the other hand, allows different functional domains in banks to easily interface with the data from others without having to share a common system, and independently of data models or formats. This form of "ad hoc" data integration would allow different communities to create separate "just-in-time" databases, each reflecting a different view/area of interest over the same data. To address the challenges of these use cases as well as many other examples stemming from today's and future applications, we clearly have to move beyond the state of the art.

1.2 Pitfalls of heterogeneity

Database architects typically define every layer of the data analysis stack a priori, having a specific use case scenario in mind. The presence of heterogeneity, however, complicates the way practitioners perform data analysis. Specifically, as depicted in Figure 1.1, different types of heterogeneity affect each step of data analysis:

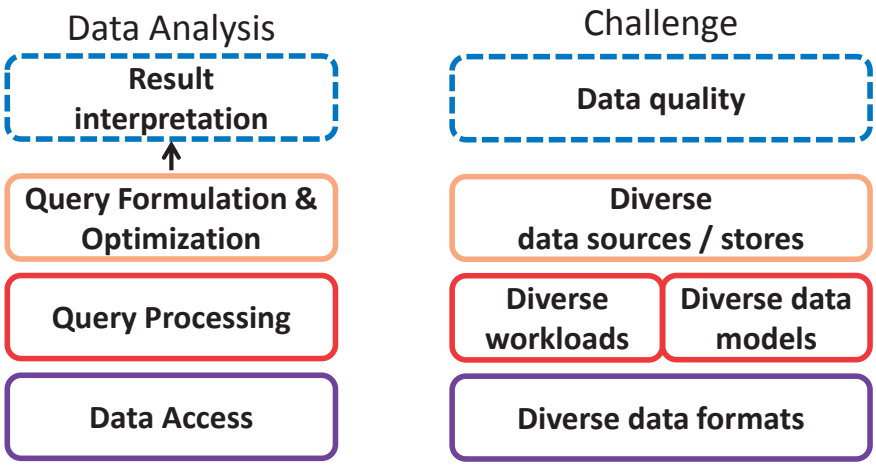


Figure 1.1 – The data analysis stack, and the heterogeneity-related challenge in each layer.

- **Data Access of Heterogeneous Formats.** Data volume has been increasing exponentially and data variety increases, with an escalating number of new formats. Still, database systems only operate efficiently over loaded data, i.e., data converted from its original raw format into the system's internal data format. As a consequence, there is a growing impedance mismatch between the original structures holding the data in the raw files and the structures used by query engines for efficient processing.

- **Query Processing over Heterogeneous Models.** Besides tabular, relational representations, practitioners model data as hierarchies, arrays, etc. Thus, evaluating queries over diverse datasets is non-trivial. In addition, practitioners launch different types of queries over the same data. For example, even if a dataset is stored in JSON representation, the analysis over it can resemble typical OLAP queries. Thus, data analysis solutions over heterogeneous data models have always involved a trade-off: be flexible and accommodate multiple diverse data models at the cost of performance, or be rigid and specialized for a specific scenario [233], thus leading users to employ a different system per use case.

- **Query Processing for Heterogeneous Workloads.** Organizations increasingly require analytics on fresh operational data to derive timely insights. To meet these requirements, database engines have to efficiently support hybrid transactional and analytical workloads (*HTAP*) over shared data. Designing a database engine that can serve mixed workloads efficiently is challenging, because besides the diverse requirements and characteristics of OLTP and OLAP workloads, the workloads negatively interfere with each other due to hardware resource contention [210].

- **Query Optimization over Heterogeneous Sources.** The typical enterprise data architecture consists of several actively updated data sources (e.g., NoSQL systems, data warehouses), and a central data lake, such as HDFS, in which all the data is periodically loaded through ETL processes. To simplify query processing and optimization, state-of-the-art data analysis approaches solely operate on top of the local, historical data in the data lake, and ignore the

fresh tail end of data that resides in the original remote sources. However, as many business operations depend on real-time analytics, this approach is no longer viable. The alternative is hand-crafting the analysis task to explicitly consider the characteristics of the various data sources and identify optimization opportunities, rendering the overall analysis convoluted.

- **Interpreting dirty data.** The process of gathering, storing, and integrating heterogeneous datasets introduces several inaccuracies in the data. For example, the presence of duplicate entries is a typical issue when integrating multiple datasets. Analysts spend 50%-80% of their time preparing dirty data before it can be used for information extraction [172]. Therefore, data cleaning is a major hurdle for data analysis.

1.3 Thesis Statement and Contributions

Heterogeneity, both in data and in query workload, significantly affects the way practitioners perform data analysis. This thesis redesigns the data analysis stack so that every layer of the stack natively caters for heterogeneity in terms of input datasets and query workloads. The end goal is decoupling the type of analysis that a user performs from the original data representation / location.

Thesis Statement

Big data is increasingly heterogeneous, nevertheless data management systems must assume homogeneity to provide efficient data analysis. As data sizes grow exponentially, data harmonization overheads are often prohibitive to business applications. To analyze heterogeneous data efficiently, data management architectures must rely on unifying abstractions to manage and mask heterogeneity, and generate customized analysis engines just-in-time to fully adapt to the use case at hand.

1.3.1 The end goal: Data virtualization

A change of paradigm is required for data analysis processes to address data diversity and volume. Database systems must become dynamic entities whose construction is lightweight and fully adaptable to the datasets and the queries. *Data virtualization* [259], i.e, abstracting data out of its form and manipulating it regardless of the way it is stored or structured, is necessary. To offer unconditional data virtualization, database systems must abolish static decisions like pre-loading data and using "pre-cooked" query operators; such operators are too generic to cope with multiple types of inputs, making them inefficient. In addition, database systems must allow users to use the query language of their choice and express data cleaning tasks declaratively, while masking heterogeneity through the use of heterogeneity-aware, high-level building blocks. In this thesis, we argue that data management must become a lightweight, flexible service, instead of a monolithic software centering around the status quo of static operators and growing obese under the weight of new requirements.

To this end, this thesis makes the following key contributions:

- **Adapting a query engine to data formats.** Traditionally, data has always had to "adapt" to the query engine of a system. We propose a reverse, novel approach: Dynamically adapting a query engine to the underlying raw data files. Thus, we introduce *JIT access paths*, which define access methods through generation of file- and query-specific scan operators, using information available at query time.

- **Customizing a query engine based on the underlying data models.** We present a system design that bridges the conflicting requirements for generality in analysis and minimal response times. This design supports both relational as well as nested data by using an expressive, optimizable query algebra that is richer than the relational one. We couple this powerful query algebra with on-demand adaptation techniques to eliminate numerous query execution overheads. To overcome the complexity of the broad algebra, we avoid the use of general-purpose abstract operators. Instead, we dynamically create an optimized engine implementation per query using code generation.

- **Customizing a query engine based on the underlying hardware.** We present the blueprint of a new architecture for designing database engines that target mixed transactional and analytical workloads. The architecture explicitly targets emerging server hardware, which incorporates accelerators like GPGPUs.

- **Optimizing queries across diverse data sources.** We show how to perform real-time analytics over a data lake ecosystem, while simultaneously masking the complexity of dealing with multiple data sources and offering fast response times. We design a data virtualization module which is pluggable to modern scale-out data processing systems and which provides a unified view over multiple systems that are heterogeneous in terms of i) data model, ii) update rates, and iii) query capabilities. Besides facilitating querying, the data virtualization module optimizes the overall analysis by considering both established cost-based query optimization techniques as well as the properties of the underlying data sources to generate an efficient execution plan.

- **Cleaning data in declarative fashion.** We address the coverage and efficiency problems of data cleaning by introducing a language that can express multiple types of cleaning operations. The language serves a purpose similar to that of SQL for data management in terms of expressivity and optimization: First, SQL allows users to manage data in an organized way and is subjective to how each user wants to manipulate the data. Similarly, data cleaning is a task that is subjective to the user's perception of cleanliness and therefore requires a language that allows users to express their requests in a simple yet efficient way. Second, SQL is backed by the highly optimizable relational calculus. Our proposed language thus is backed by an optimizable underlying representation as well, and goes through a three-level translation and optimization process; a different family of optimizations is applied at each abstraction level.

1.3.2 Thesis Roadmap

This thesis is organized as follows:

- Chapter 2 provides the necessary background on concepts we utilize and extend in the context of this thesis.
- Chapter 3 shows how a query engine can adapt to heterogeneous data formats, instead of vice versa. Specifically, the chapter introduces Just-In-Time (JIT) access paths, which are generated dynamically per file and per query instance.
- Chapter 4 presents a system design principle for analytical query engines that serve queries over data of varying models. The system design offers i) generality in analysis and ii) minimal response times. To achieve this, the design couples i) a query algebra that supports both relational and nested data with ii) on-demand customization mechanisms that collapse all layers of the system architecture at query time.
- Chapter 5 describes how to perform data analysis declaratively over numerous actively updated data sources. This chapter describes a data virtualization module that employs a location-aware compiler and a powerful two-phase optimizer, and is pluggable to scale-out computational frameworks; it supports and optimizes diverse analytics over a global virtual schema that masks data source variety and complexity.
- Chapter 6 introduces an all-purpose data cleaning query language, which models both straightforward cleaning operations, such as syntactic checks, as well as complex cleaning building blocks, such as clustering algorithms, while being naturally extensible and parallelizable.
- Chapter 7 presents the vision of Heterogeneous-HTAP (H^2TAP), a new architecture for database engines that fully utilize emerging server hardware (i.e., machines with heterogeneous parallelism) to serve both transactional and analytical workloads.
- Chapter 8 concludes the thesis and presents future directions.

2 Background

This chapter presents a brief overview of topics that are central to this thesis. Specifically, we first discuss and motivate analysis over raw data. Then, we discuss about algebras that are powerful enough to accommodate the analysis of diverse data models. We also present the state-of-the-art in terms of query execution, i.e., how numerous database systems rely on code generation and compilation to minimize query execution overheads, and how one can utilize modern hardware accelerators. Finally, we provide background on numerous data cleaning techniques.

2.1 Raw Data Analytics

Ideally, analyzing data in disparate sources would begin with ad hoc querying of the data. Instead, databases are designed to query data stored in an internal data format, which is tightly integrated with the remaining query engine and, hence, typically proprietary. Thus, if users wish to query raw data, they traditionally first load it into a database. A recent alternative to blindly loading data into a database involves asking queries directly over raw data. This section provides the necessary background on the alternative ways of accessing data.

2.1.1 Traditional: Loading and Accessing Data

Relational database systems initially load data into the database and then access it through the scan operators in the query plan. Each scan operator is responsible for reading the data belonging to a single table. Following the Volcano model [117], every call to the `next()` method of the scan operator returns a tuple or batch of tuples from the table. The scan operator in turn retrieves data from the buffer pool – an in-memory cache of disk pages.

In modern column-stores [58] the implementation details differ but the workflow is similar. A call to the `next()` method of a column-store scan operator returns a chunk of a column or the whole column. In addition, the database files are often memory-mapped, relying on the operating system's virtual memory management instead of relying on a buffer pool internal to the database.

A major overhead in this method is loading the data in the first place [36, 100]. Queries may also trigger expensive I/O requests to bring data into memory but from there on, accessing data does not entail significant overheads. For instance, a database page can be type-cast to the corresponding C/C++ structure at compile time. No additional data conversion or re-organization is needed.

2.1.2 Accessing Data through External Tables

External tables allow data in external sources to be accessed as if it were in a loaded table. External tables are usually implemented as file-format-specific scan operators. MySQL, for instance, supports external tables through its pluggable storage engine API [187]. The MySQL CSV Storage Engine returns a single tuple from a CSV file when the `next()` method is called: it reads a line of text from the file, tokenizes the line, parses the fields, converts each field to the corresponding MySQL data type based on the table schema, forms a tuple and finally passes the tuple to the query operators upstream.

The efficiency of external tables is affected by a number of factors. First, every access to a table requires tokenizing/parsing a raw file. For CSV, it requires a byte-by-byte analysis, with a set of branch conditions, which are slow to execute [59]. Second, there is a need to convert and re-organize the raw data into the data structures used by the query engine. In the case of MySQL, every field read from the file must be converted to the equivalent MySQL data type and placed in a MySQL tuple. Finally, these costs are incurred repeatedly, even if the same raw data has been read previously.

2.1.3 Querying Data *In Situ*

Abolishing the data loading phase has the potential to facilitate data exploration. At the same time, accessing data using external tables fails to leverage the effort spent by previous queries. An alternative for accessing data advocates treating "raw" data as a first-class citizen of a DBMS [36, 132, 145, 144, 197, 41, 78, 79, 219, 80, 80, 55, 255, 81, 135, 141]. The scan operators of such systems must be able to handle not only the binary data format that is understandable by the database, but also raw data. In addition, specialized data structures must facilitate raw data access by providing indexing support over raw data.

In Situ Access & Databases. The IBM Starburst [123] project on extensible relational database management systems introduces specialized access paths for external data. The key challenges in Starburst are to define the interfaces and mechanisms to expose externally-resident data, to reflect their added capabilities in the query language, and to make the database components, such as the query optimizer, aware of their costs and advantages [223].

The "NoDB philosophy" [36] advocates that in many scenarios database systems can treat raw data files as first-class citizens and operate directly over them. The implementation of NoDB [36], PostgresRaw, is a DBMS that implements techniques specifically designed to

operate over raw data. During query execution, PostgresRaw incrementally builds auxiliary indexing structures called "positional maps", which store the position of frequently-accessed fields. Future accesses to the raw data use the positional maps to skip tokenizing/parsing files, which reduces the overhead in accessing raw data.

Recent work in HyPer [185] also considers querying CSV data. Parallelizing the phases of loading and utilizing vectorization primitives enables HyPer to bulk load data at wire speed. Another alternative to "vanilla" in situ processing is invisible loading, developed for MapReduce [29] by piggybacking on MapReduce jobs. Tuples are incrementally loaded and organized into a database while data is being processed. In a related approach, Polybase [95] treats data in Hadoop clusters as external tables to a DBMS.

In Situ Access & Cloud Systems. The success of the Map-Reduce paradigm [93] led to a great number of systems operating over data stored in HDFS [226], built over Hadoop [8], or using a similar distributed runtime environment to access in situ data at scale [38, 43, 39, 54, 198]. Google Dremel [182] and Apache Drill [5] – Dremel's open-source variant – also query data in situ, with data stored in various storage layers.

In Situ Access of Scientific data. FastBit [255] is a collection of compressed bitmap indexes that enable efficient exploration of read-only scientific data. FastBit is used internally by FastQuery [81], a framework for posing selection queries over datasets in formats such as HDF5 and NetCDF [242] that has been shown to scale out. SDS/Q [55] and SCANRAW [78] perform parallel analysis over a scientific file format. For array data, Data Vaults [135] have been built on top of MonetDB [58] and offer access to repositories of external files. They are equipped with a cache manager and an optimizer for this data format, while enabling queries using SciQL, a domain specific query language.

2.2 Query Languages for Diverse Data Models

Queries targeting heterogeneous data must consider the unavoidable model heterogeneity and enable the combination of information from diverse data sources. The query language used must also enable users to "virtualize" the original data, i.e., apply powerful transformations over the output of a query. The aim of this thesis is to provide native support for non-relational data sources, therefore the relational calculus is insufficient as a base for a query language in this context.

In the past years, researchers have proposed languages and algebras for rich data models. The majority of the research efforts centered around efficient XML management [26, 27, 68, 106, 180, 203]. Another relevant line of work regarding rich data model support comes from the programming languages domain: List and monad comprehensions [69, 249] are popular constructs in (functional) programming languages, and are richer than the relational calculus; they offer support for query recursion and for arbitrary data nestings. Comprehensions have been used to iterate through collections in programming languages such as Haskell, Scala, F#, Python and JavaScript. From a database-oriented perspective, the Kleisli functional

query system [254] uses the comprehension syntax and has been used as a facilitator for data integration tasks due to its expressive power [67]. RodentStore [89] uses list comprehensions as the basis for its storage algebra; it manipulates the physical representation of the data by utilizing the expressive nature of comprehensions to express transformations. LINQ [181] exposes query comprehension syntax and enables queries over numerous databases.

The works comprising this thesis use the *monoid comprehension calculus* [104, 105], which we summarize in the rest of this section.

The monoid comprehension calculus

A *monoid* is an algebraic construct term stemming from category theory. A monoid of type T comprises an associative binary operation $\oplus$ and a zero element $\mathcal{Z}_\oplus$. The binary operation, called *merge function*, indicates how two objects of type T can be combined. The *zero element* $\mathcal{Z}_\oplus$ is the left and right identity of the merge function $\oplus$; for every object x of type T, the equivalence $\mathcal{Z}_\oplus \oplus x = x \oplus \mathcal{Z}_\oplus = x$ is satisfied.

Monoids can be used to capture operations between both primitive and collection data types. The latter also require the definition of a *unit function* $\mathcal{U}_\oplus$, which is used to construct singleton values of a collection type (e.g., a list of one element). For example, $(+,0)$ represents the primitive *sum* monoid for integer numbers. The pair $(\cup,\{\})$ along with the unit function $x \to \{x\}$ represent the *set* collection monoid.

The *monoid comprehension calculus* is used to describe operations between monoids. A monoid comprehension is an expression of the form $\oplus\{e|q_1,...,q_n\}$. The terms q_i are called *qualifiers*. Each qualifier can either be

- a *generator*, taking the form $v \leftarrow e'$, where e' is an expression producing a collection, and v is a variable that is sequentially bound to each value of said collection;
- a *filter* predicate.

The expression e is called the *head* of the comprehension, and is evaluated for each value binding produced by the generators. The evaluation results are combined using the merge function $\oplus$, called the *accumulator* of the comprehension in this context. Table 2.1, originally from [105], contains the syntactic forms of the monoid comprehension calculus.

The comprehension syntax that this thesis uses is slightly altered but equivalent to the one presented, and resembles the sequence comprehensions of Scala. The syntax we use is $for\{q_1,...,q_n\}\ yield \oplus e$. As an example, suppose that we want to pose the following SQL query counting a department's employees:

NULL	null value
c	constant
v	variable
$e.A$	record projection
$\langle A_1 = e_1, ..., A_n = e_n \rangle$	record construction
if e_1 **then** e_2 **else** e_3	if-then-else statement
e_1 op e_2	op: primitive binary function (e.g., $<, +$)
$\lambda v : \tau . e$	function abstraction
$e_1(e_2)$	function application
$\mathcal{Z}_\oplus$	zero element
$\mathcal{U}_\oplus(e)$	singleton construction
$e_1 \oplus e_2$	merging
$\oplus\{e \mid q_1, ..., q_n\}$	comprehension

Table 2.1 – The monoid comprehension calculus

```
SELECT COUNT(e.id)
FROM Employees e
JOIN Departments d ON (e.deptNo = d.id)
WHERE d.deptName = "HR"
```

The same aggregate query can be expressed in our version of comprehension syntax as follows:

```
for { e <- Employees, d <- Departments,
      e.deptNo = d.id, d.deptName = "HR"} yield sum 1
```

This example requires us to use the *sum* monoid for integers to perform the count required. Other primitive monoids we can use in our queries include, among others, *max* and *average*. Similarly, queries with universal or existential quantifiers can use the $\vee$ and $\wedge$ monoids for boolean types. More complex operations, such as top-k and bloom filters, can also be treated as monoids.

Monoid comprehensions also support nested expressions. A query requesting an employee's name along with a collection of all the departments this employee is associated with is expressed as follows:

```
for { e <- Employees, d <- Departments, e.deptNo = d.id}
yield set (emp := e.name,
           depList := for {d2 <- Departments,d.id = d2.id}
                      yield set d2)
```

From calculus to (nested) algebra

For each incoming query, the first step is translating it to a calculus expression. The calculus expression is then rewritten to an algebraic tree of a *nested relational algebra* [105]. This

Operator Name	Select	(Outer) Join	Reduce	Nest	(Outer) Unnest
Operator Symbol	$\sigma_p(X)$	$X \; \dot{\bowtie}_p \; Y$ $X \bowtie_p Y$	$\Delta_p^{\oplus/e}$	$\Gamma_{p/g}^{\oplus/e/f}$	$\dot{\mu}_p^{path}(X)$ $\mu_p^{path}(X)$

Superscript & Subscript	p: Filtering Expression	e: Output Expression
	f: Groupby Expression	g: Non-nullable Expression
	$path$: Field to unnest	$\oplus$: Output Collection/Aggregate

Table 2.2 – The operators of the nested relational algebra.

algebra resembles the relational one, and relational optimization techniques are applicable to it. On top of that, it offers first-class support for operations related to unnesting of queries over nested data. The operators of the nested relational algebra are depicted in Table 2.2. The *selection*, *join*, and *outer join* operators are identical to their relational counterparts. The subscript p represents the expression based on which filtering occurs (e.g., $x < 5$ for a selection, $t1.id = t2.id$ for a join, etc.). *Reduce* and *nest* are overloaded versions of the relational projection and the grouping operator respectively. The symbols e, f, p, and g correspond to algebraic expressions that are used during operator evaluation (e.g., f represents the expression to group results by). The symbol $\oplus$ represents the type of collection/aggregate to be output. Finally, the *unnest* and *outer unnest* operators "unroll" a collection field $path$ that is nested within an object. The subscript p indicates that an expression is used to filter out the results of this operator, and the superscript $path$ specifies which is the nested (collection) field to be "unrolled".

Expressive Power. Monoid comprehensions bear similarities to list and monad comprehensions [69, 249], which are constructs popular in functional programming languages. This thesis opts for monoid comprehensions as a "wrapping" layer for a variety of languages because they allow inputs of different types (e.g., sets and arrays) to be used in the same query. Query results can also be "virtualized" to the layout/ collection type requested; different applications may require different representations for the same results. Examples include, among others, representing the same original data either as a matrix or by using a relational-like tabular representation, and exporting results as bag collections while the original inputs are lists. This capability aids in "virtualizing" the original data as per the user's needs.

Crucially, a query language based on monoid comprehensions lends itself perfectly to translation to other languages. Support for a variety of query languages can be provided through a "syntactic sugar" translation layer, which maps queries written in the original language to the internal notation. Doing so enables users to formulate queries in their language of choice. Specifically, monoid comprehensions are a theoretical model behind XQuery's FLWOR expressions, and also an intermediate form for the translation of OQL [105]. The monoid comprehension calculus is also sufficient to express relational SQL queries. SPARQL queries over data representing graphs can also be mapped to the monoid comprehensions calculus [87].

2.3 Analytical Query Processing

The works in this thesis heavily rely on runtime code generation to accelerate query execution. This section thus discusses different query execution techniques, before focusing on runtime code generation.

The Volcano iterator model. When a query is posed in a database system, it is generally processed by a query planner / optimizer, resulting in an algebraic plan. This plan, expressed in the form of a tree, is traditionally interpreted using the Volcano iterator model [117]. Every operator of the plan exposes a general API, consisting of *open()*, *next()* and *close()* function calls. Whenever an operator's *next()* method is called, a request for a new tuple is sent to the operator's children operators. While being a simple and intuitive interface, its very generality actually penalizes performance. The fact that the *next()* function will be called for every tuple leads to increased costs, considering that function calls will take place even for very simple operations. In addition, as these function calls are typically virtual, they lead to frequent branch mispredictions. Finally, the constant changes in control flow lead to poor code locality.

Block-oriented query processing. To address these performance concerns, approaches have appeared suggesting block-oriented query processing, i.e. generating more that one tuple with every *next()* call of an operator ([201, 58, 59]). For such approaches, the costs resulting from the multiple calls of functions are significantly reduced. In addition, this type of processing also allows exploitation of modern hardware, such as vectorized execution by using SIMD instructions. On the other hand, by processing blocks, the output of operators needs to be materialized before being provided as input to a subsequent operator. Thus, the pipelining capabilities of the iterator model are no longer fully exploited, leading to increased memory bandwidth consumption.

Runtime code generation. In an attempt to keep the best of both worlds, namely both pipelining capabilities and better utilization of modern hardware without sustaining interpretation overheads, HyPer [190] introduces a novel execution model, based on the following guidelines:

- The query execution needs to be data-centric instead of operator-centric. Execution revolves around data that are kept in the CPU registers as long as possible, even if it means deviating from the traditional operator model [117].
- By using push-based execution instead of the traditional pull-based Volcano execution [117], the resulting query engine benefits from better code and data locality.
- Compile queries to low-level machine code that is optimized to fully exploit modern hardware.

Query compilation (i.e., runtime code generation) results in minimal code, with the majority of work taking place in "tight loops" over tuples. Such a code pattern facilitates prefetching and accurate branch prediction. In addition, the code generated contains fewer branches than the ones encountered in static systems, given that the information known at compile time

about a query can be injected into the generated code. The alternative would be "interpreting" the query plan during query evaluation, and complicating the control flow.

In general, the use of just-in-time code generation to answer database queries has re-gained popularity, years after its initial application in System R [73]. Recently, code generation has been realized using highly efficient code templates and dynamically instantiating them to create query- and hardware-specific code [159]. HyPer [190], Impala [250], and Tupleware [88] employ the LLVM JIT compiler infrastructure [162] to generate and compile code. Other approaches have a high-level language as a starting point to generate code for queries [153, 188, 213, 224, 91]. A prominent example of the other side of low-level code generation, LegoBase [153] advocates "abstraction without regret" and staged compilation; its query engine and its optimization rules are both written in the high-level language Scala. Different optimizations can be applied in every query translation step from the original Scala representation to the C code that is eventually generated. Finally, hybrid storage layouts can also benefit by applying code generation to increase CPU efficiency [206] or to adapt the data layout at runtime [37].

2.4 Query Processing on Emerging Server Hardware

The hardware landscape exhibits two major trends to which the data management sector must adapt, namely, the generalization of GPGPUs and the specialization of multisocket CPUs. This section presents the characteristics of emerging hardware and their mismatch with modern database engines.

2.4.1 Generalization of GPGPUs

Traditionally, GPGPUs suffered from two major limitations. First, applications that used GPGPUs had to manage host (CPU) and device (GPU) memory separately, thus complicating programmability. Second, GPU device memory capacity was too limited to store all data. Therefore, applications had to manually copy data from system to device memory via the slow PCIe bus before executing a computation on the GPU. As a result, despite work that showed that GPGPUs can provide substantial improvement in performance over CPUs [62, 96, 127, 128, 258], they were not widely used in the industry because analytical queries running on GPGPUs spent most of their time transferring data. As Table 2.3 shows, however, GPGPUs are evolving from memory-limited accelerators for niche domains to general-purpose processors with radical improvements along the dimensions of performance, interfacing, and programmability[1].

Performance. The latest Pascal GPUs offer 16× higher processing power and 13.3× more memory capacity than their Tesla counterparts. GTX 1080 Ti will have an order of magnitude more cores and 4× higher memory bandwidth than even state-of-the-art multi-core CPUs. Furthermore, GPU cards which are customized for compute acceleration typically pack 2× more memory capacity and processing power over these consumer-grade graphics cards.

[1] While this section uses NVIDIA terminology, all concepts apply to AMD GPGPUs as well.

GPU	Architecture	Cores	FP32 Power (GFlops)	Mem cap (MB)	Mem b/w (GB/s)	I/f type	I/f b/w (GB/s)
GeForce 8800	Tesla	128	345.6	768	103.7	PCIe 1.0	4
GTX 580	Fermi	512	1581.1	1536	192.3	PCIe 2.0	8
GTX 780 Ti	Kepler	2304	3976.7	3072	288.4	PCIe 3.0	16
GTX 980 Ti	Maxwell	2816	5632	6144	336	PCIe 3.0	16
GTX 1080 Ti	Pascal	3328	10696	10240	400	NVLink	80-200

Table 2.3 – Processing power, memory capacity, and interconnection bandwidth of consumer-grade NVIDIA graphics cards across generations

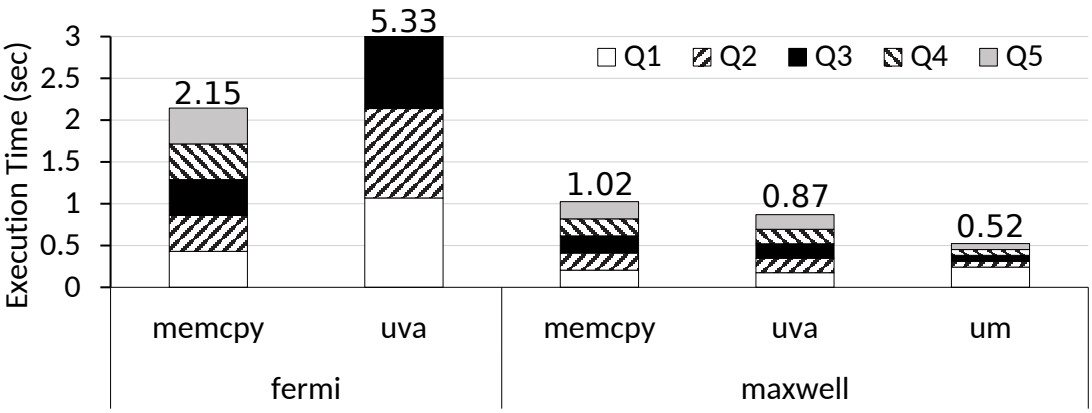


Figure 2.1 – Scan execution time under Fermi/Maxwell GPUs.

Interfacing. PCIe 3.0 already offers 4× higher bandwidth compared to PCIe 1.0, and PCIe 4.0 is expected to double the bandwidth again. In addition, NVIDIA has recently announced NVLink [193], an energy-efficient, high-bandwidth GPU-CPU or GPU-GPU interconnect that will offer at least 5× the bandwidth of the current PCIe 3.0 bus. NVLink is already being used to interconnect IBM Power CPUs and NVIDIA GPUs in Summit and Sierra, two supercomputers commissioned by the U.S DoE [195].

Programmability. Since CUDA 4.0, NVIDIA Fermi GPUs have supported Unified Virtual Addressing (UVA) [192], which enables GPUs and CPUs to share a single address space. The Kepler architecture added support for Unified Memory (UM) in CUDA 6.0 [192]. UM enables applications to offload memory management entirely to the CUDA runtime, which tracks memory accesses and migrates data to host or device memory depending on the access patterns to improve locality. With CUDA 8.0, the Pascal architecture extends UM with support for virtual memory-based page faulting in GPU; data allocated on the CPU is automatically faulted in and moved to the GPU one page at a time, only when accessed. Applications can thus oversubscribe GPU memory, i.e., allocate a chunk of memory larger than the GPU device capacity and access it using the same address pointer across CPUs and GPUs.

Figure 2.1 quantifies the net effect of some of these improvements by showing the results from a microbenchmark that executes five filter queries over a 2GB column of integers using an

M2090 Fermi GPU and a GTX 980 Maxwell GPU. Each query launches a *kernel* – a function that all the threads of a GPU device execute in parallel. The three cases in the graph present scenarios where memory is allocated separately on device and host, requiring an explicit copy operation ("memcpy"), or memory is allocated using UVA/UM and requires no copying. In the memcpy case, we report the total time taken to perform the host-to-device input copy, kernel execution, and device-to-host output copy. For UVA and UM, we report the time to execute the kernel.

There are four important observations to be made: First, the memcpy case shows a 2× improvement because of the improvement in bandwidth from 8 GB/s under PCIe 2.0 (Fermi) to 16 GB/s under PCIe 3.0 (Maxwell). Second, while UVA was 2.5× slower than memcpy under Fermi, it is 1.18× faster under Maxwell, indicating that UVA enables efficient CPU–GPU data sharing. Third, under UM, the first query takes 0.24 seconds and is 1.5× slower than under UVA. The remaining queries, however, execute in 0.07 seconds, and are 2.5× faster. After the first query, the CUDA runtime migrates the input array allocated in UM over to the GPU. Thus, subsequent queries are unaffected by the PCIe bandwidth limitation. This performance improvement required no programming effort and shows the locality benefit of using UM. Finally, comparing Fermi UVA and Maxwell UVA/UM, execution time gets a 6× reduction with UM, and a 2.46× reduction with UVA. All these speedups are noteworthy because i) they purely stem from improvements in interfaces and programmability, as the kernel does little computation, and ii) they show that efficient CPU–GPU data sharing is possible.

2.4.2 Specialization of CPUs

In stark contrast to the generalization of GPGPUs is the increasing specialization of commodity multi-socket multi-cores. An aspect of specialization which is particularly relevant to database designers is the design of hardware cache coherence (CC). All widely used multi-cores provide CC-shared memory to ensure that memory store operations performed by one core are visible to load operations performed by another core despite multiple levels of caching. CC also forms the framework for features like atomics on shared memory words and Hardware Transactional Memory. However, as the number of cores increases, the cost and complexity needed to maintain coherence across all core-private caches is also increasing dramatically [178, 257]. Research has also shown that CC presents scalability challenges for latency-sensitive workloads [178, 183].

While designing scalable CC protocols for multi-cores continues to be a challenging topic [176], hardware vendors have started investigating alternative multi-core architectures that vary widely with respect to CC support. The latest Haswell processors support three modes of CC; choosing the right mode impacts the latency and bandwidth of both core-to-core data transfers and memory accesses [183]. SoCs like TI OMAP4, OMAP5, and Samsung Exynos, which are based on the ARM v8 specification, group cores into multiple domains such that coherence is maintained within but not across domains. Intel SCC [131] is a 48-core processor that provides

non-CC shared memory with core-to-core message passing capability. IBM Cell Broadband Engine [121] is a single-chip multiprocessor with eight non-CC Synergistic Processor Elements (SPE) optimized for data processing. Given such variation among processors in providing CC, several researchers argue that system-wide CC may no longer be available in the near future [49, 50, 169], and are building software such as operating systems [47, 49, 169, 253], file systems [133], memory management libraries [50], and runtime libraries [70, 163], explicitly targeted at emerging non-CC systems.

2.5 Data Cleaning

Cleaning dirty data has been an omnipresent data management challenge. This section surveys i) frequently-required data cleaning operations, and ii) data cleaning frameworks and techniques [25, 83, 113, 134, 139, 149, 212, 235, 246] used to perform said cleaning operations.

2.5.1 Data Cleaning Operations

In the following we describe a set of popular data cleaning operations.

Denial Constraints (DC). The family of *denial constraints* [102] contains universally quantified first order language sentences that represent data dependencies, such as functional dependencies (FD) and conditional functional dependencies. DCs have the following form: $\forall t_1, ..., t_k \neg(p(x_1) \wedge p(x_2) \wedge ... p(x_n))$. If a dataset contains one or more tuples for which the predicates $p(x_1)...p(x_n)$ hold, it is considered to be inconsistent.

Duplicate Elimination. *Duplicate elimination* involves the discovery of tuples that refer to the same real-world entity [158]. The most straightforward way to detect similar tuples is a self-join that discovers identical tuples. A lighter duplicate detection form is to consider an attribute or a set of attributes that should be unique; if two tuples have the same values for that particular set of attributes, then they are considered to be duplicates. A more challenging scenario involves the case where a dataset does not contain completely identical pairs of tuples/attribute sets, but might contain *similar* pairs. In this case, the self-join predicate needs to calculate similarity instead of equality, and thus requires the user to choose an appropriate similarity metric.

Transformations & Term Validation. *Transformations* involve applying a formula to a set of values, or mapping values to a set of semantically related values [25], such as mapping a column with airport names to the corresponding cities. Semantic transformations are challenging because they require consulting auxiliary data. *Term validation* is a popular category of semantic transformations: It focuses on detecting values that are seemingly correct, but fail to adhere to a specific terminology because of, for example, a misspelling. A common technique for detecting misspellings is using a dictionary for validation. The dictionary can be, among others, a dictionary of english words or scientific terms, or even the result of a query to a portal with geographic data.

2.5.2 Data Cleaning Systems & Techniques

In the following we survey multiple categories of data cleaning techniques.

Interactive Data Cleaning. Potter's Wheel [212], OpenRefine [246], and Trifacta – the commercial version of Data Wrangler [139] – are established interactive data cleaning systems. Potter's Wheel [212] provides an interface via which the user gradually repairs her dataset. The user performs transformations, such as merging columns, and at the same time, a background daemon detects potential syntactic errors. For the daemon to detect any errors, a user has to specify patterns to which values must adhere, a set of domains to which data entries must belong, and the constraints of each domain. Pentaho [17], Knime [13] and Paxata [16] allow for more complex operations, which they express with the use of black-box user-defined-functions (UDFs).

(Semi-)Automatic Cleaning. Besides interactive cleaning toolkits, other systems attempt to detect and repair data errors automatically, asking a human for guidance when necessary. DataXFormer [25] tackles semantic transformations, such as mapping a column containing company names to a column with the stock symbols of those companies, by exploiting information from the Web or from mapping tables. Tamr [235] focuses on repairing data duplicates. Tamr maps records to corresponding blocks, and then trains classifiers within each block that decide whether a pair of records corresponds to a duplicate, a non-duplicate, or a possible duplicate. Finally, Tamr produces groups of values that represent the same entity, and suggests a representative value per group using rules and feedback from experts. Dedoop [154] allows specifying entity resolution workflows through a web-based interface and then translates them into MapReduce jobs. Each Dedoop operator is a standalone, black-box UDF. SampleClean [252] and Wisteria [122] extract a sample out of a dataset, employ users to clean it, and utilize this sample to answer aggregate queries; their focus is on data transformations, deduplication, and denial constraints.

NADEEF [82, 90] manages a set of denial constraints, and tries to update erroneous values in a way that all the rules are satisfied [82]. BigDansing [149] ports the insights of NADEEF in a distributed setting by extending MapReduce-like frameworks with support for duplicate elimination and denial constraints. BigDansing takes as input a dirty dataset along with a quality rule that is either declarative or has the form of a UDF. Then, BigDansing detects and repairs violations in a scale-out fashion. BigDansing performs a nuber of logical-level optimizations over input cleaning scripts; the optimizations focus on projection push down and blocking in the case of functional dependencies. BigDansing also focuses on physical-level optimizations, such as offering a custom join implementation. Specifically, since denial constraints often involve inequality joins, BigDansing provides a custom theta join operator.

Declarative Cleaning. The `FUSE BY` [56] operator is an extension of SQL that resolves duplicates by allowing various conflict resolution strategies, such as choosing the most common value or preferring one source over another. `FRAQL` [221] follows a similar approach by providing SQL extensions that allow transformations, duplicate elimination, and outlier detection.

All conflict resolution operations in FRAQL are expressed as standalone, opaque UDFs. QuERy [40] integrates deduplication with query processing by focusing on the optimizations that allow cleaning only the parts of the data that are needed by a given query. Ajax [110] separates the logical and physical level of the data cleaning process. At the logical level, Ajax uses a data flow graph to represent the steps of a cleaning operation. Then, at the physical level, each logical operator gets translated into an optimized implementation. Like FRAQL, Ajax provides a UDF for each operator, and therefore treats each data cleaning task as a black box.

Quantitative Data Cleaning (QDC). QDC [51, 92] discovers the best data repairing strategy using statistical methods, such as the cost of each strategy, the quality of the resulting dataset, and the statistical distortion against the original dataset. QDC focuses on discovering the optimal repair method given a set of detected errors. Statistics are also employed to measure the accuracy of error detection methods and how each method behaves in the existence of multiple types of errors; whether a method fails to detect an error due to the presence of another type of error [52]. Finally, the authors of [209] combine statistics with qualitative methods to perform data cleaning.

SQL for cleaning. SQL can express some cleaning tasks, e.g., the ones that correspond to first order logic statements [102]. SQL, however, is overall inappropriate and insufficient for data cleaning: First, SQL lacks first-class support for rich data types (e.g., JSON); one might need to convert a dataset to another format in order to clean it. A change in the intended format can be inconvenient for the user, or might complicate the cleaning process, e.g., flattening a dataset can increase data volume. In addition, relational algebra – the backend of SQL – lacks first-class support for operations from the machine learning and data mining domains.

It typically takes a combination of vanilla SQL, UDFs, extra operators, and external programs to express rich operations in SQL [84]. UDFs, however, increase complexity; each UDF appears as a black-box to the system optimizer, which is unable to optimize the entire task as a whole. Adding extra operators in the database core [204] requires coding in an operator per algorithm, which is a tedious process. As for frameworks such as Spark [260], which support both relational and iterative processing, they apply only relational optimizations [43]. The reason is that the "relational part" of Spark is engineered similarly to a DBMS with columnar storage and is equipped with an optimizer, whereas the "procedural part" executes arbirary code over BLOB-like data (RDDs [260]). Given the split Spark architecture, the Spark SQL Catalyst optimizer treats the procedural parts of an analysis script as black boxes. In summary, both for traditional RDBMS and modern scale-out frameworks, while a relational optimizer can perform rewrites based on the physical properties of the extra operators, it is non-trivial to reason about them on an algebraic level, because they fall outside of the relational logic based on which the system has been engineered.

In conclusion, SQL is designed to manipulate relational data, and is unable to express domain-specific optimizations required for data cleaning.

3 Just-in-time Access Paths

Database systems deliver impressive performance for large classes of workloads as the result of decades of research into optimizing database engines. High performance, however, is achieved at the cost of versatility. In particular, database systems only operate efficiently over loaded data, i.e., data converted from its original raw format into the system's internal data format. At the same time, data volume continues to increase exponentially and data varies increasingly, with an escalating number of new formats. The consequence is a growing impedance mismatch between the original structures holding the data in the raw files and the structures used by query engines for efficient processing. In an ideal scenario, the query engine would seamlessly adapt itself to the data and ensure efficient query processing regardless of the input data formats, optimizing itself to each instance of a file and of a query by leveraging information available at query time. Today's systems, however, force data to adapt to the query engine during data loading.

This chapter proposes adapting the query engine to the formats of raw data. It presents RAW, a prototype query engine that enables querying heterogeneous data sources transparently. RAW employs Just-In-Time (JIT) access paths, which efficiently couple heterogeneous raw files to the query engine and reduce the overheads of traditional general-purpose scan operators.

3.1 Introduction

Over the past decades, database query engines have been heavily optimized to handle a variety of workloads to cover the needs of different communities and disciplines. What is common in every case is that regardless of the original format of the data to be processed, top performance requires data to be pre-loaded: Database systems always require the original user's data to be reformatted into new data structures that are exclusively owned and managed by the query engine. These structures are typically called database pages and store tuples from a table in a database-specific format. The layout of pages is hard-coded deep into the database kernel and co-designed with the data processing operators for efficiency. Therefore, this efficiency was achieved at the cost of versatility; keeping data in the original files was not an option.

Two trends that now challenge the traditional design of database systems are the increased *variety* of input data formats and the exponential growth of the *volume* of data, both of which belong in the "Vs of Big Data" [160]. Both trends imply that a modern database system has to load and restructure increasingly variable, exponentially growing data, likely stored in multiple data formats, before the database system can be used to answer queries. The drawbacks of this process are that i) the "pre-querying" steps are a major bottleneck for users who want to quickly access their data or perform data exploration, and ii) databases have exclusive ownership over their ingested data; once data has been loaded, external analysis tools cannot be used over it any more unless data is duplicated.

Flexible and efficient access to heterogeneous raw data remains an open problem. NoDB [36] advocates *in situ* query processing of raw data and introduces techniques to eliminate data loading by accessing data in its original format and location. However, the root cause of the problem is still not addressed; there is an impedance mismatch, i.e., a costly adaptation step due to differences between the structure of the original user's data and the data structures used by the query engine. To resolve the mismatch, the implementation of NoDB relies on file- and query-agnostic scan operators, which introduce interpretation overhead due to their general-purpose nature. It also uses techniques and special indexing structures that target textual flat files, such as CSV. As its design is hard-coded to CSV files, it cannot be extended to support file formats with different characteristics (such as ROOT [63]) in a straightforward way. Finally, NoDB may import unneeded raw data while populating caches with recently accessed data. Therefore, even when accessed in situ as in the case of NoDB, at some moment, data must always "adapt" to the query engine of the system.

In this chapter, we propose a reverse, novel approach. We introduce *RAW*, a flexible query engine that *dynamically adapts* to the underlying raw data files and to the queries themselves, rather than adapting data to the query engine. In the ideal scenario, the impedance mismatch between the structure in which data is stored by the user and by the query engine must be resolved by having the query engine seamlessly adapt itself to the data, thus ensuring efficient query processing regardless of the input data formats. RAW creates its internal structures at runtime and defines the execution path based on the query requirements. To bridge the impedance mismatch between the raw data and the query engine, RAW introduces *Just-In-Time (JIT) access paths* and *column shreds*. Both methods build upon in situ query processing [36], column-store engines [59] and code generation techniques [159] to enable efficient processing of heterogeneous raw data. To achieve efficient processing, RAW delays work to be done until it has sufficient information to reduce the work's cost, enabling one to access and combine diverse datasets without sacrificing performance.

JIT access paths define access methods through generation of file- and query-specific scan operators, using information available at query time. A JIT access path is dynamically-generated, removing overheads of traditional scan operators. Specifically, multiple branches are eliminated from the critical path of execution by coding information such as the schema or data type conversion functions directly into each scan operator instance, enabling efficient execution.

The flexibility that JIT access paths offer also facilitates the use of query processing strategies such as column shreds. We introduce column shreds to reduce overheads that cannot be eliminated even with JIT access paths. RAW creates column shreds by pushing scan operators *up the query plan*. This tactic ensures that a field (or fields) is only retrieved after filters or joins to other fields have been applied. Reads of individual data elements and creation of data structures are delayed until they are actually needed, thus creating only subsets (*shreds*) of columns for some of the raw data fields. The result is avoiding unneeded reads and their associated costs. Column shreds thus efficiently couple raw data access with a columnar execution model.

Motivating Example. The ATLAS Experiment of the Large Hadron Collider at CERN stores over 140 PB of scientific data in the ROOT file format [63]. Physicists write custom C++ programs to analyze this data, combining them with other secondary data sources, such as CSV files. Some of the analysis implies complex calculations and modelling, which is impractical on a relational database system. The remaining analysis, however, requires simple analytical queries, e.g., building a histogram of "events of interest" with a particular set of muons, electrons or jets. A DBMS is desirable for this latter class of analysis because declarative queries are significantly easier to express, to validate and to optimize compared to a C++ program. Loading, i.e., replicating, 140 PB of data into a database, however, would be cumbersome and costly. Storing this data at creation time in a database would constrain the use of existing analysis tools, which rely on specific file formats. Therefore, a query engine that queries the raw data directly is the most desirable solution. To process ROOT and be useful in practice, a system must have performance competitive to that of the existing C++ code. RAW, our prototype system, outperforms handwritten C++ programs by two orders of magnitude. RAW adapts itself to the ROOT and CSV file formats through code generation techniques, enabling operators to work over raw files as if they were the native database file format.

Contributions. Our contributions are as follows:

- We design a query engine which adapts to raw data file formats and not vice versa. Based on this design, we implement a data- and query-adaptive engine, RAW, that enables querying heterogeneous raw data efficiently.
- We introduce Just-In-Time (JIT) access paths, which are generated dynamically per file and per query instance. Besides offering flexibility, JIT access paths address the overheads of existing scan operators for raw data. JIT access paths are 1.3× to 2× faster than state-of-the-art methods [36].
- We introduce column shreds, a novel execution method over raw data to reduce data structure creation costs. With judicious use of column shreds, RAW achieves an additional 6× speedup for highly selective queries over CSV files; for a binary format, it approaches the performance of a traditional DBMS with fully-loaded data. Column shreds target a set of irreducible overheads when accessing raw data (e.g., data conversion). In our experiments these reach up to 80% of the query execution time.

- We apply RAW in a real-world scenario that cannot be accommodated by a DBMS. RAW enables the transparent querying of heterogeneous data sources, while outperforming the existing hand-written approach by two orders of magnitude.

Outline. The rest of this chapter is structured as follows: Section 3.2 reviews existing methods to access in situ data. Section 3.3 briefly describes RAW, our prototype query engine. Section 3.4 introduces Just-In-Time access paths. Section 3.5 introduces column shreds. Sections 3.4 and 3.5 also evaluate our techniques through a set of experiments. Section 3.6 evaluates a real-world scenario enabled through the application of our approach. Section 3.7 concludes the chapter.

3.2 Preliminaries: Accessing Data through Positional Maps

Positional maps are data structures that the implementation of NoDB [36] uses to optimize in situ querying. They are created and maintained dynamically during query execution to track the (byte) positions of data in raw files. Positional maps, unlike traditional database indexes, index the *structure* of the data and not the actual data, reducing the costs of tokenizing and parsing raw data sources.

Positional maps work as follows: When reading a CSV file for the first time, the scan operator populates a positional map with the byte location of each attribute of interest. If the attribute of interest is in column 2, then the positional map will store the byte location of the data in column 2 for every row. If the CSV file is queried a second time for column 2, there is no need to tokenize/parse the file. Instead, the positional map is consulted and we jump to that byte location. If the second query requests a different column, e.g., column 4, the positional map is still used. The parser jumps to column 2, and incrementally parses the file until it reaches column 4. The positional maps involve a trade-off between the number of positions to track and future benefits from reduced tokenizing/parsing.

Positional maps outperform external tables by reducing or eliminating tokenizing and parsing, yet still lead to a number of inefficiencies. First, positional maps carry a significant overhead for file formats where the location of each data element is known deterministically, such as cases when the location of every data element can be determined from the schema of the data. For instance, the FITS file format, widely-used in astronomy, stores fields in a serialized binary representation, where each field is of fixed size. Additionally, there are costs we cannot avoid despite using positional maps, such as the costs of creating data structures and converting data to populate them with. For every data element, the scan operator needs to check its data type in the database catalog and apply the appropriate data type conversion.

3.3 The RAW Query Engine

RAW is a prototype query engine that adapts itself to the input data formats and queries, instead of forcing data to adapt to it through a loading process. RAW offers file format-agnostic querying without sacrificing performance. To achieve this flexibility, it applies in situ query processing, columnar query execution and code generation techniques in a novel query engine design. The design can be extended to support additional file formats by adding appropriate file-format-specific plug-ins. Because RAW focuses on the processing of read-only and append-like workloads, it follows a columnar execution model, which has been shown to outperform traditional row-stores for read-only analytical queries [24, 58, 233, 234], and exploits vectorized columnar processing to achieve better utilization of CPU data caches [59]. Additionally, it applies code generation techniques to generate query access paths on demand, based on the input data formats and queries.

RAW Internals. We have built RAW on top of Google's Supersonic library of relational operators for efficient columnar data processing [116]. The Supersonic library provides operators that apply cache-aware algorithms, SIMD instructions, and vectorized execution to minimize query execution time. Supersonic does not, however, have a built-in data storage manager. RAW extends the functionality of Supersonic to enable efficient queries over raw data by i) generating data format- and query-specific scan operators, and ii) enabling scan operators to be pushed higher in the produced query plan, thus avoiding unnecessary raw data accesses. A typical physical query plan, therefore, consists of the scan operators of RAW for accessing the raw data and the Supersonic relational operators.

RAW creates two types of data structures to speed-up queries over files. For textual data formats (e.g., CSV), RAW generates positional maps to assist in navigating through the raw files. In addition, RAW preserves a pool of column shreds populated as a side-effect of evaluating previous similar queries, to reduce the cost of re-accessing the raw data. RAW considers these position and data caches for each incoming query when selecting an access path.

Catalog and Access Abstractions. Each file exposed to RAW is given a name (can be thought of as a table name). RAW maintains a catalog with information about raw data file instances such as the original filename, the file format, and the corresponding relational schema. RAW accepts partial schema information (i.e., the user may declare only fields of interest instead of declaring thousands of fields) for file formats that allow direct navigation based on an attribute name, instead of navigation based on the binary offsets of fields. As an example, for ROOT data, we could store the schema of a ROOT file as *((“ID”,INT64), (“el_eta”,FLOAT), (“el_medium”,INT32))* if only these fields were to be queried, and ignore the rest 6 to 12 thousand fields in the file. For each “table”, RAW keeps the types of accesses available for its corresponding file format, which are mapped to the generic access paths abstractions understood by the query executor, i.e., sequential and index-based scans. For example, there are scientific file formats (e.g., ROOT) for which a file corresponds to multiple tables, as objects in a file may contain lists of sub-objects. These sub-objects are accessible using the identifier of their parent. For such file

types, RAW maps this id-based access to an index-based scan. Enhancing RAW with support for additional file formats simply requires establishing mappings for said formats.

Physical Plan Creation. The logical plan of an incoming query is file-agnostic, and consists of traditional relational operators. As a first step, we consult the catalog of RAW to identify the files corresponding to tables in the plan's scan operators. RAW converts the logical query plan to a physical one by considering the mappings previously specified between access path abstractions and concrete file access capabilities. We also check for available cached column shreds and positional maps (if applicable to the file format). Then, based on the fields required, we specify how each field will be retrieved. For example, for a CSV file, potential methods include i) straightforward parsing of the raw file, ii) direct access via a positional map, iii) navigating to a nearby position via a positional map and then performing some additional parsing, or iv) using a cached column shred. Based on these decisions, we split the field reading tasks among a number of scan operators to be created, each assigned with reading a different set of fields, and push some of them higher in the plan. To push scan operators higher in the plan instead of traditionally placing them at the bottom, we extend Supersonic with a "placeholder" generic operator. RAW can insert this operator at any place in a physical plan, and use it as a placeholder to attach a generated scan operator. Code generation enables creating such custom efficient operators based on the query needs.

Creating Access Paths Just In Time. Once RAW makes all decisions for the physical query plan form, it creates scan operators on demand using code generation. First, RAW consults a template cache to determine whether this specific access path has been requested before. If not, a file-format-specific plug-in is activated for each scan operator specification, which turns the abstract description into a file-, schema- and query-aware operator. The operator specification provided to the code generation plug-in includes all relevant information captured from the catalog and the query requirements. Depending on the file format, a plug-in is equipped with a number of methods that can be used to access a file, ranging from methods to scan fields from a CSV file (e.g., `readNextField()`), up to methods acting as the interface to a library that is used to access a scientific data format, as in the case of ROOT (e.g., `readROOTField(fieldName, id)`).

Based on the query, appropriate calls to plug-in methods are put together per scan operator, and this combination of calls forms the operator, which is compiled on the fly. The freshly-compiled library is dynamically loaded into RAW and the scan operators are used as the leaves of the remaining query plan / tree. The library is also registered in the template cache to be reused later in case a similar query is submitted. The generated scan operators traverse the raw data, convert the raw values, and populate columns.

RAW supports code-generated access paths for CSV, flat binary, and ROOT files. Adding access paths for additional file formats is straightforward due to the flexible architecture of RAW. Sections 3.4 and 3.5 describe how RAW benefits from JIT access paths for raw data of different formats and how it avoids unnecessary accesses to raw data elements, respectively.

3.4 Adapting to raw data

Just-In-Time (JIT) access paths are a new method for a database system to access raw data of heterogeneous file formats. We design and introduce JIT access paths in RAW to dynamically adapt to raw datasets and to incoming queries. JIT access paths are an enabler for workloads that cannot be accommodated by traditional DBMS, due to i) the variety of file formats in the involved datasets, ii) the size of the datasets, and iii) the inability to use existing tools over the data once they have been loaded. In the rest of this section, we present JIT access paths and evaluate their performance.

3.4.1 Just-In-Time Access Paths

JIT access paths are generated dynamically for a given file format and a user query. Their efficiency is based on the observation that some of the overheads in accessing raw data are due to the general-purpose design of the scan operators used. Therefore, customizing a scan operator at runtime to specific file formats and queries partially eliminates these overheads.

For example, when reading a CSV file, the data type of the column being currently read determines the data conversion function to use. Mechanisms to implement data type conversion include a pointer to the conversion function or a switch statement. The second case can be expressed in pseudo-code as follows:

```
FILE* file
int column       // current column

for every column {
  char *raw      // raw data
  Datum *datum   // loaded data
  //read field from file
  raw = readNextFieldFromFile(file)

  switch (schemaDataType[column])
    case IntType: datum = convertToInteger(raw); break;
    case FloatType: datum = convertToFloat(raw); break;
    ...
}
```

The `switch` statement and `for` loop introduce branches in the code, which significantly affect performance [190]. Even worse, both are in the critical path of execution. As the data types are known in advance, the `for` loop and the `switch` statement can be unrolled. Unrolled code executes faster because it causes fewer branches.

Opportunities for Code Generation. JIT access paths eliminate a number of overheads of general-purpose scan operators. The opportunities for code generation optimizations vary depending on the specificities of the file format. For example:

- Unrolling of columns, i.e., handling each requested column separately instead of using a generic loop, is appropriate for file formats with fields stored in sequence, forming a tuple. Each unrolled step can be specialized based on, for example, the datatype of the field.

- For some data formats, the positions of fields can be deterministically computed, and therefore we can navigate for free in the file by injecting the appropriate binary offsets in the code of the access paths, or by making the appropriate API calls to a library providing access to the file (as in the case of ROOT).

- File types such as HDF [237] and shapefile [101] incorporate indexes over their contents, B-Trees and R-Trees respectively. Indexes like these can be exploited by the generated access paths to speed-up accesses to the raw data.

- For hierarchical data formats, a JIT scan operator coupled with a query engine supporting a nested data model could be used to maintain the inherent nesting of some fields, or flatten some others, based on the requirements of the subsequent query operators. These requirements could be based on criteria such as whether a nested field is projected by the query (and therefore maintaining the nesting is beneficial), or just used in a selection and does not have to be recreated at the query output.

Generally, for complex file formats, there are more options to access data from a raw file. Our requirement for multiple scan operators per raw file, each reading an arbitrary number of fields, further increases the complexity. Traditional scan operators would need to be too generic to support all possible cases. Code generation in the context of JIT access paths enables us to create scan operators on demand, fine-tuning them to realize the preferred option, and to couple each of them with the columnar operators for the rest of query evaluation. As we will see in Section 3.5, this flexible transition facilitates the use of methods like column shreds.

Example. Consider a query that scans a table stored in a CSV file. The file is being read for the first time; thus, a positional map is built while the file is being parsed. Compared to a general-purpose CSV scan operator, the generated operator includes the following optimizations:

- *Column loop is unrolled.* Typically, a general-purpose CSV scan operator, such as a scan operator of the NoDB implementation or of the MySQL CSV storage engine, has a `for` loop that keeps track of the current column being parsed. The current column is used to verify a set of conditions, such as "if the current column must be stored in the positional map, then store its position". In a general-purpose in situ columnar execution, another condition would be "if the current column is requested by the query plan, then read its value". In practice, however, the schema of the file is known in advance. The actions to perform per column are also known. Thus, the column loop and its inner set of `if` statements can be unrolled.

- *Data type conversions built into the scan operator.* A general-purpose scan operator needs to check the data type of every field being read in a metadata catalog. As the schema is known, it can be coded into the scan operator code, as illustrated earlier.

More specifically, for a memory-mapped CSV file with 3 fields of types *(int, int, float)*, with a positional map for the 2nd column and a query requesting the 1st and 2nd fields, the generated pseudo-code for the example query is the following:

```
FILE *file
while (!eof) {
  Datum *datum1, *datum2 // values read from fields 1,2

  raw = readNextFieldFromFile(file)
  datum1 = convertToInteger(raw)

  addToPositionalMap(currentPosition)

  raw = readNextFieldFromFile(file)
  datum2 = convertToInteger(raw)

  skipFieldFromFile()

  CreateTuple(datum1, datum2)
}
```

For this query, the scan operator reads the first field of the current row. It converts the raw value just read to an integer and also stores the value of the file's position indicator in the positional map. The operator then reads the next (2nd) field of the row, also converting it to an integer. Because we do not need to process the 3rd field, we skip it, and create a result for the row examined. The process continues until we reach the end of file. The generated pseudo-code for a second query requesting the 2nd and 3rd columns is the following:

```
for (every position in PositionalMap) {
  Datum *datum2, *datum3   // values read from fields 2,3

  jumpToFilePosition(position)

  raw = readNextFieldFromFile(file)
  datum2 = convertToInteger(raw)

  raw = readNextFieldFromFile(file)
  datum3 = convertToFloat(raw)

  CreateTuple(datum2, datum3)
}
```

Improving the Positional Map. Positional maps reduce the overhead of parsing raw files [36] but add significant overhead for file formats where the position of each data element can be determined in advance. JIT access paths eliminate the need for a positional map in such cases. Instead, a function is created in the generated code that resolves the byte position of the data element directly by computing its location. For instance, for a binary file format where every tuple is of size `tupleSize` and every data element within it is of size `dataSize`, the location of the 3rd column of row 15 can be computed as `15*tupleSize + 2*dataSize`. The result of the formula is directly included in the generated code. Different file formats also benefit from different implementations of the positional map; an example is presented in Section 3.6.

3.4.2 Evaluating raw data access strategies

File formats vary widely, and each format benefits differently from JIT access paths. We examine two file formats that are representative of two "extreme" cases. The first is CSV, a text-based format where attributes are separated by delimiters, i.e., the location of column N varies for each row and therefore cannot be determined in advance. The second is a custom binary format where each attribute is serialized from its corresponding C representation. For this specific custom format, we exploit the fact that the location of every data element is known in advance because every field is stored in a fixed-size number of bytes. The plug-in for this format includes methods to either i) read specific datatypes from a file, without having to convert this data, or ii) skip a binary offset in a file. The same dataset is used to generate the CSV and the binary file, corresponding to a table with 30 columns of type integer and 100 million rows. Its values are distributed randomly between 0 and 10^9. Being integers, the length of each field varies in the CSV representation, while it is fixed-size in the binary format.

The sizes of the raw CSV and binary files are 28GB and 12GB respectively. The experiments are run on a dual socket Intel Xeon, described in the first row of Table 3.1. The operating system is Red Hat Enterprise Linux Server 6.3 with kernel version 2.6.32. The compiler used is GCC 4.4.7 (with flags -msse4 -O3 -ftree-vectorize -march=native -mtune=native). The files are memory-mapped. The first query runs over cold caches. Intermediate query results are cached and available for re-use by subsequent queries.

Machine	Description
Xeon Dual-Socket	2 x Intel Xeon CPU E5-2660 @ 2.20GHz, 8 cores/CPU 128GB RAM RAID-0 of 7 250 GB 7500 RPM SATA 64KB L1 cache (32KB L1d, 32KB L1i) per core 256KB L2 cache per core; 20MB L3 shared cache
Xeon Octo-Socket	8 x Intel Xeon CPU E7-28867 @ 2.13GHz, 10 cores/CPU 192GB RAM 1TB 7200 RPM SAS HDD 64KB L1 cache (32KB L1d, 32KB L1i) per core 256KB L2 cache per core; 30MB L3 shared cache

Table 3.1 – Hardware setup for experiments evaluating RAW.

We run the microbenchmarks in RAW. The code generation is done by issuing C++ code through a layer of C++ macros.

Data Loading vs. In Situ Query Processing. The following experiment compares different techniques, all implemented in RAW, for querying raw data to establish the trade-off between in situ query processing and traditional data loading. "DBMS" corresponds to the behavior of a column-store DBMS, where all raw data is loaded before submitting the first query. The data loading time of the DBMS is included as part of the first query. "External Tables" queries the raw file from scratch for every query. "In Situ" is our implementation of NoDB [36] over RAW, where access paths are *not* code-generated. "JIT" corresponds to JIT access paths.

The workload comprises two queries submitted in sequence. The two queries are the following:

```
SELECT MAX(col1) WHERE col1 < [X]
SELECT MAX(col11) WHERE col1 < [X]
```

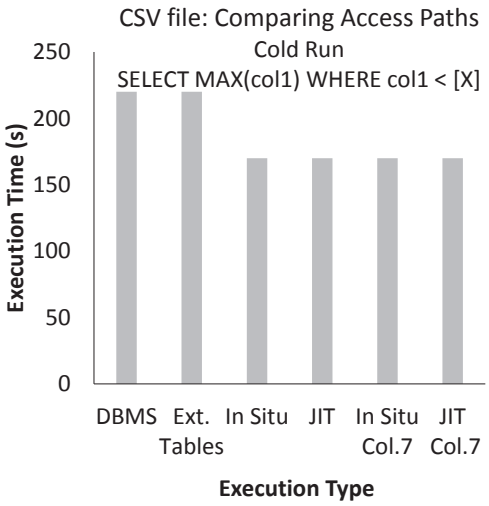
We report results for different selectivities by changing the value of X.

The first experiment queries a CSV file. "In Situ" and "JIT" both utilize positional maps, which are built during the execution of the first query and used in the second query to locate any missing columns. Because different policies for building positional maps are known to affect query performance [36], we test two different heuristics. The first populates the positional map every 10 columns; i.e., it tracks positions of columns 1, 11, 21, etc. The second populates the positional map every 7 columns.

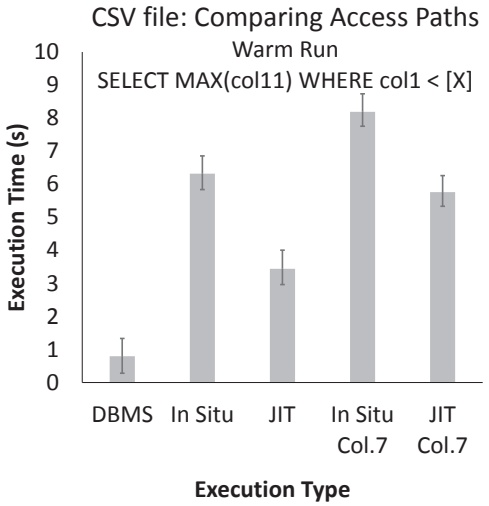
Figure 3.1a depicts the results for the first query (cold file system caches). The response time is approximately 220 seconds for "DBMS" and "External Tables" and 170 seconds for "In Situ" and "JIT". "DBMS" and "External Tables" do the same amount of work for the first query, building an in-memory table with all data in the file before executing the query. "In Situ" and "JIT" do fewer data conversions and populate fewer columns (only those actually used by the query), which reduces the execution time. In the case of JIT access paths, the time to generate and compile the access path code is included in the execution time of the first query, contributing approximately 2 seconds. In both cases, however, I/O dominates the response time and the benefit of JIT access paths is not particularly visible (except that the compilation time is amortized).

For the second query, the results are depicted in Figure 3.1b. We vary selectivity from 1% to 100% and depict the average response time, as well as deltas for lowest and highest response time. The execution time for "External Tables" is an order of magnitude slower, thus it is not shown. The "In Situ" and "JIT" cases use the positional map to jump to the data in column 11. The variations "In Situ - Column 7" and "JIT - Column 7" need to parse incrementally from the nearest known position (column 7) to the desired column (column 11). In all cases, a custom version of `atoi()`, the function used to convert strings to integers, is used as the length of the string is stored in the positional map. Despite these features, "DBMS" is faster, since data is already loaded into the columnar structures used by the query engine, whereas the "JIT" case spends approximately 80% of its execution on accessing raw data. It is important to note, however, that the extra loading time incurred by the "DBMS" during the first query may not be amortized by fast upcoming queries; these results corroborate the observations of the NoDB work [36].

Comparing "In Situ" with "JIT", we observe that the code generation version is approximately 2× faster. This difference stems from the simpler code path in the generated code. The "In Situ - Column 7" and "JIT - Column 7" techniques are slower as expected compared to their counterparts that query the mapped column 11 directly, due to the incremental parsing that needs to take place.



(a) Raw data access is faster than loading (I/O masks part of the difference).



(b) "DBMS" is faster, as all needed data is already loaded. JIT access paths are faster than general-purpose in situ.

Figure 3.1 – JIT access paths vs. In Situ and DBMS approaches: Cold and warm run of a query over CSV data.

We now turn to the binary file. No positional map is necessary now. The "In Situ" version computes the positions of data elements during query execution. The "JIT" version hard-codes the positions of data elements into the generated code. For the first query, both "In Situ" and "JIT" take 70 seconds. The "DBMS" case takes 98 seconds. I/O again masks the differences between the three cases. The results for the second query are shown in Figure 3.2. The trends of all cases are similar to the CSV experiment. The performance gaps are smaller because no data conversions take place.

JIT access paths breakdown. To confirm the root cause of speedup in the "JIT" case, we profile the system using VTune[1]. We use the same CSV dataset as before, and ask the query `SELECT MAX(col1) WHERE col1 <[X]` on a warm system. Figure 3.3 shows the comparison of the "JIT" and "In Situ" cases for a case with 40% selectivity. Unrolling the main loop, simplifying the parsing code and the data type conversion reduces the costs of accessing raw data. Populating columns and parsing the file remain expensive though. In the next section we introduce column shreds to reduce these costs.

Summary. JIT access paths significantly reduce the overhead of in situ query processing. For CSV files and for a custom-made binary format, JIT access paths are up to 2× faster than traditional in situ query processing techniques. Traditional in situ query processing, adapted to columnar execution, is affected by the general- purpose and query-agnostic nature of the scan operators that access raw data. Just-In-Time code generation, however, introduces

[1] http://software.intel.com/en-us/intel-vtune-amplifier-xe

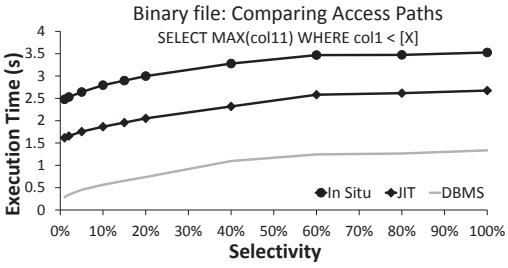


Figure 3.2 – JIT access paths vs. In Situ and DBMS approaches: For binary files, JIT access paths outperform traditional in situ query processing.

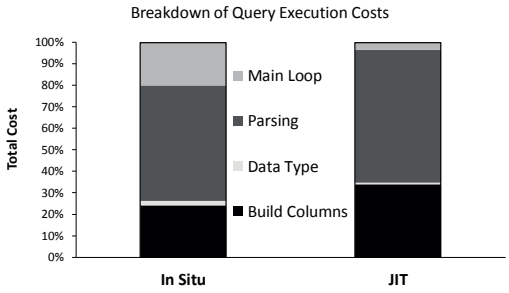


Figure 3.3 – Benefits from adapting to data: Unrolling the main loop, simplifying parsing and data type conversions reduce the time spent "preparing" raw data.

a compilation overhead, incurred the first time a specific query is asked. Two methods to address this issue are i) maintaining a "cache" of libraries generated as a side-effect of previous queries, and re-using when applicable (RAW follows such an approach), and ii) using a JIT compiler framework, such as LLVM [162], which can reduce compilation times [190].

As we see in the next section, the flexibility and efficiency offered by JIT access paths combined with column shreds will enable us to further increase the performance of RAW.

3.5 When To Load Data

JIT access paths reduce the cost of accessing raw data. There are, however, inherent costs with raw data access that cannot be removed despite the use of JIT access paths. These costs include i) multiple raw data accesses, ii) converting data from the file format (e.g., text) to the database format (e.g., C types), and iii) creating data structures to place the converted data.

Use of column shreds is a novel approach that further reduces the cost of accessing raw data. So far, we have been considering the traditional scenario in which we have one scan operator per file, reading the fields required to answer a query and building columns of values. Column shreds build upon the flexibility offered by JIT scan operators. Specifically, we can generate multiple operators for a single data source, each reading an arbitrary subset of fields in a row-wise manner from the file. Our aim is to have each operator read the minimum amount of data required at the time. To achieve this, based on when a field is used by a query operator (e.g., it is used in a join predicate), we place the scan operator reading the field values *higher in the query plan*, in hope that many results will have been filtered out by the time the operator is launched. As a result, instead of creating columns containing all the values of a raw file's requested fields, we end up creating *shreds* of the columns.

In the rest of this section, we present column shreds and evaluate their behavior. We consider the applicability of using column shreds in different scenarios, gauge their effects and isolate the criteria indicating when they should be applied.

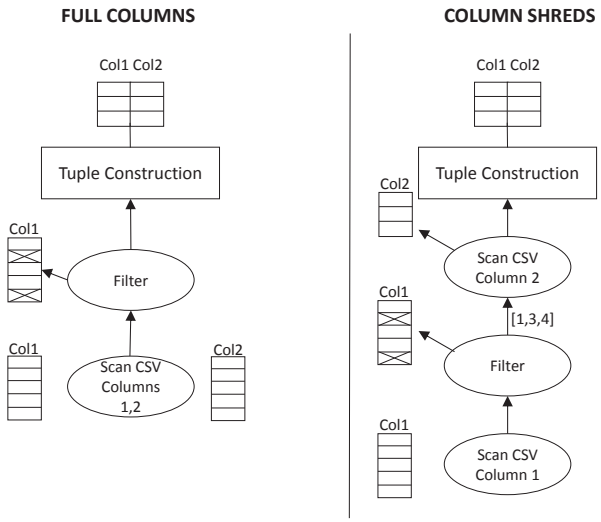


Figure 3.4 – "Full columns" vs. "Column Shreds". "Full columns": all column values are pre-loaded into columnar structures. "Column shreds": column pieces are only built as needed: in the example, Col2 is only loaded with the rows that passed the filter condition on Col1.

3.5.1 Shredding Columns

Creating entire columns at startup is a conceptually simple approach. A small experiment, however, illustrates the potential overhead it carries. Consider the query `SELECT MAX(col2) FROM table WHERE col1 < N`. The number of entries from col2 that need to be processed to compute the `MAX` depends on the selectivity of the predicate on col1. If columns 1 and 2 are entirely loaded, in what we now call *"full columns"*, then some elements of column 2 will be loaded but never used. If the selectivity of the predicate is 5%, then 95% of the entries read from column 2 will be unnecessary for the query. This is an undesirable situation, as time is spent on creating data structures and loading them with data that is potentially never needed but still expensive to load.

The "column shreds" approach dictates creating and populating columns with data only when that data is strictly needed. In the previous example, we load only the entries of column 2 that qualify, i.e., if the selectivity of the predicate is 5%, then only 5% of the entries for column 2 are loaded, greatly reducing raw data accesses.

Figure 3.4 illustrates the difference between the two column creation strategies. In the case of full columns, a single scan operator populates all required columns. For this example, column shreds are implemented by generating a columnar scan operator for column 2 and pushing it *up the query plan*. In addition, the (Just-In-Time) scan operators are modified to take as input the identifiers of qualifying rows from which values should be read. In Figure 3.4, this is the set of rows that pass the filter condition. For CSV files, this *selection vector* [59] actually contains the closest known binary position for each value needed, as obtained from the positional map. The remaining query plan and operators are not modified.

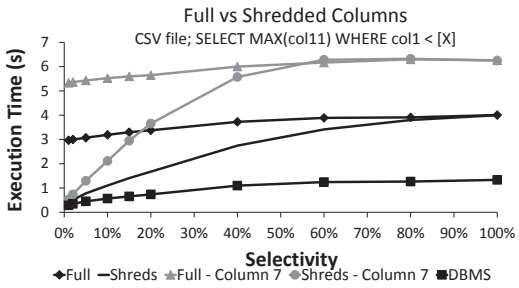


Figure 3.5 – "Full columns" vs. "Column Shreds" - CSV. For the 2nd query over a CSV file, column shreds are always faster or exactly the same as full columns, as only elements of column 11 that pass the predicate are loaded from the file.

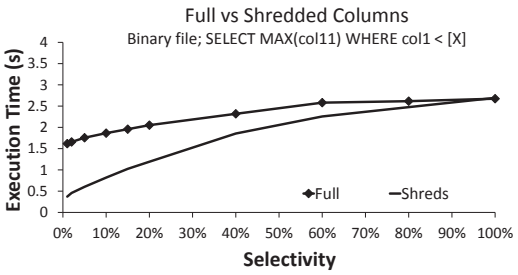


Figure 3.6 – "Full columns" vs. "Column Shreds" - Binary. For the 2nd query over a binary file, we see the same behavior as for CSV: use of column shreds is always faster than use of full columns or exactly the same for 100% selectivity.

It is important for the multiple scan operators accessing a file to work in unison. For the majority of file formats, reading a field's values from a file requires reading a file page containing unneeded data. Therefore, when a page of the raw file is brought in memory due to an operator's request, we want to extract all necessary information from it and avoid having to re-fetch it later. Our operators accept and produce vectors of values as input and output. After a scan operator has fetched a page and filled a vector with some of the page's contents, it forwards the vector higher in the query tree. Generally, at the time a subsequent scan operator requests the same file page to fill additional vectors, the page is still "hot" in memory, so we do not incur I/O again. If we had opted for operators accepting full columns, we would not have avoided duplicate I/O requests for pages of very large files.

RAW maintains a pool of previously created column shreds. A shred is used by an upcoming query if the values it contains subsume the values requested. The replacement policy we use for this cache is LRU. Handling the increasing number of varying-length shreds after a large number of queries and fully integrating their use can introduce bookkeeping overheads. Efficient techniques to handle this issue can be derived by considering query recycling of intermediate results, as applied in column stores [136, 189].

3.5.2 Full Columns vs. Column Shreds

To evaluate the behavior of column shreds, we compare them with the traditional "full columns" approach. The hardware and workload used are the same as in Section 3.4. We use simple analytical queries of varying selectivity so that the effect of full vs shredded columns is easily quantifiable, instead of being mixed with other effects in the query execution time. All cases use JIT access paths. For CSV files, a positional map is built while running the first query and used for the second query. As in Section 3.4, we include two variations of the positional map: one where the positional map tracks the position of a column requested by the second query, and one where the positional map tracks a nearby position.

System	File Format	Execution Time (s)
DBMS	CSV	380 s
Full Columns	CSV	216 s
Column Shreds	CSV	216 s
DBMS	Binary	42 s
Full Columns	Binary	22 s
Column Shreds	Binary	22 s

Table 3.2 – Execution time of the 1st query over a table with 120 columns of integers and floating-point numbers. A traditional DBMS is significantly slower in the 1st query due to data loading.

The execution time of the first query is not shown because there is no difference between full and shredded columns: in both cases, every element of column 1 has to be read. Figure 3.5 shows the execution time for the second query over the CSV file of 30 columns and 100 million rows. For lower selectivities, column shreds are significantly faster ($\sim 6\times$) than full columns, because only the elements of column 11 that pass the predicate on column 1 are read from the raw file. Compared to the traditional in situ approach evaluated in Section 3.4, the improvement reaches $\sim 12\times$. As the selectivity increases, the behavior of column shreds converges to that of full columns. Column shreds are always better than full columns, or exactly the same for 100% selectivity. When incremental parsing is needed, then data is uniformly more expensive to access. In all cases, the extra work in the aggregator operator, which has more data to aggregate as the selectivity increases, contributes to the gradual increase in execution time. Compared to the DBMS case, however, the increase in response time for full and shredded columns is steeper. The reason is that reading the file and aggregating data are done at the same time and both actions interfere with each other. For binary files, the same behavior is observed (Figure 3.6). Although no data conversion takes place, the other loading-related costs, e.g., populating columns, still affect the "full columns" case.

The next set of experiments uses files with wider tables (more columns) and more data types, including floating-point numbers. There are now 120 columns in each file and 30 million rows. The sizes of the CSV and binary files are 45GB and 14GB respectively. In the traditional DBMS case, all columns in the file are created before launching queries. In the "full columns" case, all columns needed by the query are created as the first step of a query. In the "column shreds" case, columns are only created when needed by some operator. In the "DBMS" case, the loading time is included in the execution time of the first query. Column 1, with the predicate condition, is an integer as before. The column being aggregated is now a floating-point number, which carries a greater data type conversion cost. The queries and remaining experimental setup are the same as before.

Table 3.2 shows the execution times for the first query. For CSV files, although I/O masks a significant part of the cost, the DBMS is 164 seconds slower, as it loads (and converts) all columns in advance, even those not part of subsequent queries. Full and shredded columns

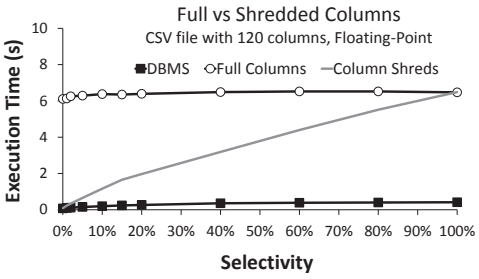


Figure 3.7 – "Full columns" vs. "Column Shreds". CSV files with floating-point numbers carry a higher data type conversion cost. The DBMS case is significantly faster.

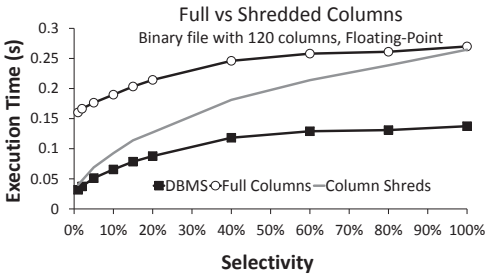


Figure 3.8 – "Full columns" vs. "Column Shreds". The binary format requires no conversions, so the absolute difference between DBMS and column shreds is very small.

are the same for the first query, as the entire column must be read to answer it. For binary files, the first query is nearly 2× slower for the DBMS. Interestingly, we may also compare the CSV and binary file formats directly. Both hold the same data, just in different representations. Querying CSV is significantly slower due to the higher cost of converting raw data into floating-point numbers and the larger file size.

The execution times for the second query in the case of the CSV file are shown in Figure 3.7. Using column shreds is competitive with "DBMS" only for lower selectivities. The curve gets steeper due to the higher cost of converting raw data into floating-point numbers.

In the binary case (Figure 3.8), there is no need for data type conversions. Therefore, the use of column shreds is competitive with the DBMS case for a wider range of selectivities. It is approximately 2× slower for 100% selectivities, yet the absolute time differences are small. The slowdown is due to building the in-memory columnar structures, and could only be resolved if the entire set of database operators could operate directly over raw data.

3.5.3 Column Shreds Tradeoffs

So far, we examined simple analytical queries with the goal of isolating the effects of shredding columns of raw data. Intuitively, postponing work as long as possible in the hope that it can be avoided appears to be always of benefit. In this section, we examine whether this assumption is true for other types of queries.

Speculative Column Shreds

For some file formats, the strict form of using scan operators to create column shreds for a single field each time may not be desirable. For example, when reading a field from a file, it may be comparatively cheap to read nearby fields. If these nearby fields are also needed by the query - e.g., they are part of a predicate selection to be executed upstream - then it may be preferable to *speculatively* read them to reduce access costs (e.g., parsing).

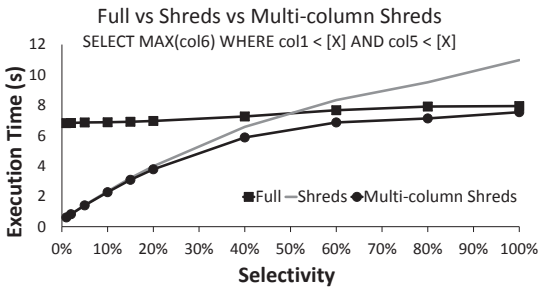


Figure 3.9 – Shredding Policies: Creating shreds of requested nearby columns in one step is beneficial when accessing raw data in multiple steps is costly.

In the next experiment we ask the query `SELECT MAX(col6) FROM file1 WHERE file1.col1 <[X] AND file1.col5 <[X]` over a CSV file. A positional map already exists (for columns 1 and 10), and the data for column 1 has been cached by a previous query. We compare the following three cases:

- full columns for fields 5 and 6 (column 1 is already cached)
- a column shred for field 5 (after predicate on field 1) and a column shred for field 6 (after predicate on field 5)
- column shreds for fields 5 and 6 after predicate on column 1 (i.e., *"multi-column shreds"*) using a single operator

As depicted in Figure 3.9, for selectivities up to 40%, creating one column shred each time is faster because we process less data. After this point, the parsing costs begin to dominate and override any benefit. The intermediate case, however, provides the best of both cases: if we speculatively create the column shred for field 6 at the same time as the one for field 5, the tokenizing/parsing cost is very small. Pushing the scan operator for field 6 higher means that the system loses "locality" while reading raw data.

Column Shreds and Joins

Column shreds can benefit for queries with joins, too. For some file formats, however, we must consider where to place the scan operator. Intuitively, columns to be projected after the join operator should be created on demand as well. That is, the join condition would filter some elements and the new columns to be projected would only be populated with those elements of interest that passed the join condition. In practice, there is an additional effect to consider, and in certain scenarios it is advantageous to create such a column before the join operator.

When considering hash joins, the right-hand side of the join is used to build a hashtable. The left-hand side probes this hashtable in a pipelined fashion. The materialized result of the join includes the qualifying probe-side tuples in their original order, along with the matches in the hashtable.

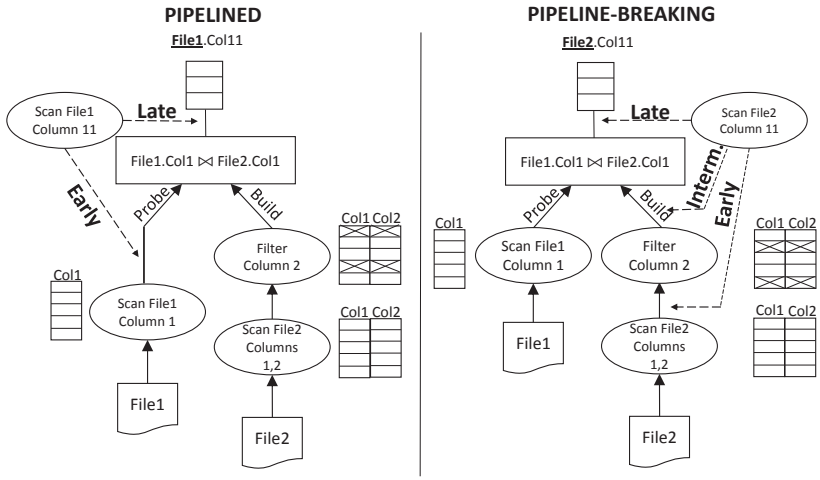


Figure 3.10 – Shredding Policies: Possible points of column population based on join side.

Let us consider the following query over two CSV files:

```
SELECT MAX(col11) FROM file1, file2
WHERE file1.col1=file2.col1 AND file2.col2 < [X]
```

Both file1 and file2 contain the same data, but file2 has been shuffled. We examine the cases in which an additional column to be projected belongs to file1 (left-hand side of the join) or to file2 (right-hand side of the join). We assume that column 1 of file1 and columns 1 and 2 of file2 have been loaded by previous queries, to isolate the direct cost of each case. We change X to alter the number of rows from file2 participating in the join.

Both cases are shown in Figure 3.10. The "Pipelined" case corresponds to retrieving the projected column from file1 and the "Pipeline Breaking" to retrieving it from file2. Both cases have two common points in the query plan where the column to be projected can be created; these are called "Early" and "Late" in Figure 3.10. The "Early" case is before the join operator (i.e., full columns); the "Late" case is after (i.e., column shreds). In the "Pipeline-Breaking" scenario, we also identify the "Intermediate" case, where we push the scan of the projected column after having applied all selection predicates, yet before applying the join. The result is creating shreds that may carry some redundant values.

The first experiment examines the "Pipelined" case. Two copies of the original CSV dataset with 100 million rows are used. The second copy is shuffled. The results are shown in Figure 3.11, also including the default "DBMS" execution for reference. The behavior is similar to that of full vs. shredded columns for selection queries: column shreds outperform full columns when selectivity is low, and the two approaches converge as selectivity increases. The reason of convergence is that the ordering of the output tuples of the join operator follows the order of entries in file1. The pipeline is not broken: therefore, the scan operator for column 11, which is executed (pipelined) after the join operator, reads the qualifying entries via the positional map in sequential order from file1. We also notice that for complex operations, such as joins,

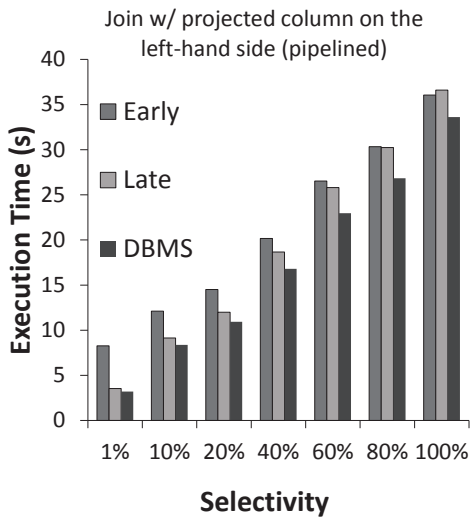


Figure 3.11 – Shredding Policies: If the column to be projected is on the "pipelined" side of the join, then delaying its creation is a better option.

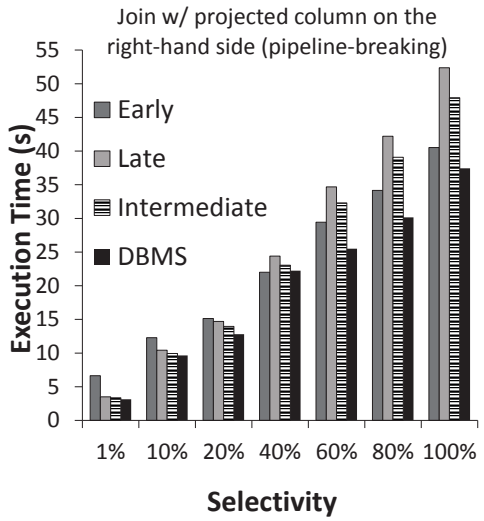


Figure 3.12 – Shredding Policies: If the projected column is on the "breaking" side, picking its point of creation depends on the join selectivity.

the fact that we access raw data is almost entirely masked due to the cost of the operation itself and the use of column shreds. For small selectivities, we observe little difference.

The second experiment examines the remaining case, which we call "Pipeline-breaking". The column to be projected is now from file2. The results are shown in Figure 3.12. DBMS, full and shredded columns perform worse than their pipelining counterparts. As the selectivity of the query increases, the performance of column shreds deteriorates, eventually becoming worse than full columns. The intermediate case exhibits similar behavior, but is not as heavily penalized for high selectivities as the late case. The reason for this behavior is the non-sequential memory accesses when reading the data. In the "DBMS" and "full columns" cases, column values are not retrieved in order, as they have been shuffled by the join operation. Even worse, in the case of column shreds it is the byte positions of the raw values stored in the positional map that have been shuffled. This leads to random accesses to the file (or to the memory-mapped region of the file). Pages loaded from the file, which already contain lots of data not needed for the query (as opposed to tight columns in the case of "full columns"), may have to be read multiple times during the query to retrieve all relevant values. This sub-optimal access pattern ends up overriding any benefits obtained from accessing a subset of column 11 in the case of column shreds.

To confirm this behavior, we use the *perf* [18] performance analyzing tool to measure the number of DTLB misses in the "pipeline-breaking" scenario. We examine the two "extreme" cases for an instance of the query with 60% selectivity. Indeed, the "full columns" case has 900 million DTLB misses and 1 billion LLC misses, while the "column shreds" case has 1.1 billion DTLB misses and 1.1 billion LLC misses due to the random accesses to the raw data.

Summary. The use of column shreds is an intuitive strategy that can provide performance gains for both selection queries and joins, where the gains are a function of query selectivity. Column shreds, however, cannot be applied naively, as loading data without considering locality effects can increase the per-attribute reading cost. In such cases of higher selectivity, multi-column shreds for selections and full creation of newly projected columns that break the join pipeline for joins provide the best behavior in our experiments.

3.6 Use Case: The Higgs Boson

The benchmarks presented in the previous sections demonstrate that JIT access paths combined with column shreds can reduce the costs of querying raw data. In practice, however, the impact of these methods depends on the specificities of each file format. Because we cannot possibly evaluate our techniques with the multitude of file formats and workloads in widespread use, we instead identify one challenging real-world scenario where data is stored in raw files and where DBMS-like query capabilities are desirable.

The ATLAS experiment [22] at CERN manages over 140 PB of data. ATLAS is not using a DBMS because of two non-functional requirements, namely i) the lifetime of the experiment: data should remain accessible for many decades; therefore, vendor lock-in is a problem, and ii) the dataset size and its associated cost: storing over 140 PB in a DBMS is a non-trivial, expensive task. Specifically, for a DBMS to serve the ATLAS experiment, the contents contained in the ROOT files first have to be converted into a tabular representation. Then, loading the data is a significant investment both in time and resources that requires duplicating ROOT data in a vendor-specific data format. In addition, ROOT files contain thousands of attributes. Relational databases generally apply much lower restrictions on the number of columns that a table can contain. For example, PostgreSQL allows for 250-1600 columns per table depending on the data type, while DB2 allows for 500-1012 columns per table. SQL Server's "wide tables" allow for 30000 columns per table, but only if the data is very sparse (i.e., the contents of the table are mostly null values); the maximum size in bytes of a wide table row remains the same as in traditional SQL Server tables. Finally, as loading the whole dataset is a complex task, physical partitioning of the original data can be required.

The ATLAS experiment built a custom data analysis infrastructure instead of using a traditional DBMS. At its core is the ROOT framework [63], widely used in high-energy physics, which includes its own file format and provides a rich data model with support for table-like structures, arrays or trees. ROOT stores data in a variety of layouts, including a columnar layout with optional use of compression. The framework also includes libraries to serialize C++ objects to disk, handles I/O operations transparently, and implements an in-memory "buffer pool" of commonly-accessed objects.

To analyze data, ATLAS physicists write custom C++ programs, extensively using ROOT libraries. Each such program "implements" a query, which typically consists of reading C++ objects stored in a ROOT file, filtering its attributes, reading and filtering nested objects, projecting attributes of interest, and usually aggregating the final results into a histogram. ROOT does not provide declarative querying capabilities; instead, users code directly in C++, using ROOT to manage a buffer pool of C++ objects transparently.

In an ideal scenario, physicists would write queries in a declarative query language, such as SQL. Queries are easier to express in a declarative query language for the average user. Query optimization also becomes possible, with the query engine determining the most appropriate way to execute the query.

We implement a query of the ATLAS experiment ("Find the Higgs Boson") in RAW to test the real-world applicability of querying raw data based on JIT access paths and column shreds. The JIT access paths in RAW emit code that calls the ROOT I/O API, instead of emitting code that directly interprets the bytes of the ROOT format on disk. The emitted code calls ROOT's `getEntry()` method to read a field instead of parsing the raw bytes, as the ROOT format is complex and creating a general-purpose code generator for ROOT would have been beyond the scope of this work.

ROOT is a binary format where the location of every attribute is known or can be computed in advance. Therefore, processing ROOT files does not require a positional map. Instead, the code generation step queries the ROOT library for internal ROOT-specific identifiers that uniquely identify each attribute. These identifiers are placed into the generated code. In practice, the JIT access path knows the location and can access each data element directly. We utilize the ROOT I/O API to generate scan operators that are performing identifier-based accesses (e.g., leading to the call of readROOTField(name,10) for a field's entry with ID equal to 10), thus pushing some filtering downwards, avoiding full scans and touching less data.

For this experiment, each ATLAS ROOT file contains information for a set of events, where an event is an observation of a collision of two highly energized particles. The Higgs query filters events where the muons, jets, and electrons in each event pass a set of conditions, and where each event contains a given number of muons/jets/electrons. In the hand-written version, an event, muon, jet, or electron is represented as a C++ class. A ROOT file contains a list of events, i.e., a list of C++ objects of type event, each containing within a list of C++ objects for its corresponding muons, jets, and electrons. In RAW, these are modelled as the tables depicted in Figure 3.14. Therefore, the query in RAW goes through the following steps:

- Reading CSV data to obtain the numbers of "good runs" of the experiment along with other useful information (**GoodRuns_CSV**).

- Filtering ROOT data concerning events. A disjunctive predicate over 6 fields is used. The result is joined with the information obtained from the CSV data (**GoodEvents**).

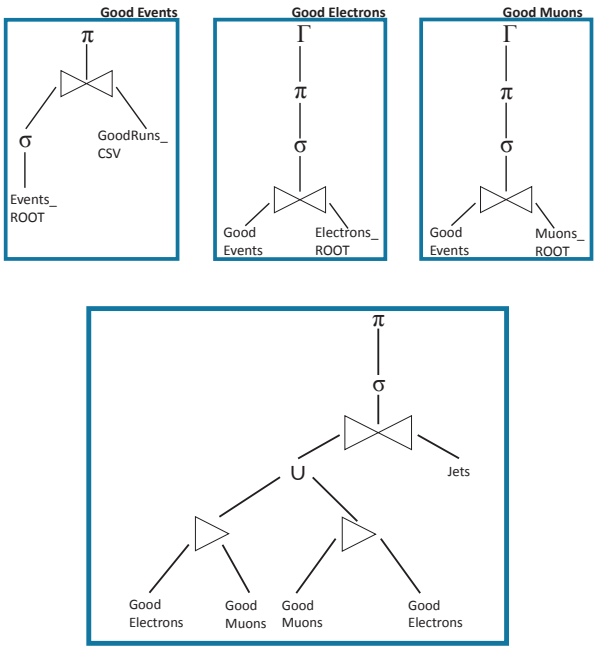


Figure 3.13 – Simplified version of the ROOT query plan. The overall query is depicted in steps.

- Joining the qualifying events with ROOT data concerning electrons, performing additional filtering using both conjunctive and disjunctive predicates involving 5 fields, and calculating an aggregate (**GoodElectrons**).
- Joining the qualifying events with ROOT data concerning muons, performing additional filtering using both conjunctive and disjunctive predicates involving >10 fields, and calculating an aggregate (**GoodMuons**).
- Performing two anti-joins between the previous two results.
- Computing the union of the two anti-joins, and performing a join with ROOT data concerning jets. Finally, the query filters the result; the remaining events are the Higgs candidates.

A simplified version of the overall query plan is depicted in Figure 3.13.

The dataset used is stored in 127 ROOT files, totaling 900 GB of data. Additionally, there is a CSV file representing a table, which contains the numbers of the "good runs", i.e., the events detected by the ATLAS detector that were later determined to be valid. Traditionally, a separate DBMS would maintain this list of "good runs". RAW, however, transparently queries and joins data in different file formats, so the CSV file with "good runs" is queried directly and joined with the ROOT files. The experiments are run on an octo socket Intel Xeon (Table 3.1) using the same operating system and compiler as before. We use a single core as each event is processed independently. The number of cores does not change the behavior of either system. In practice, events would be partitioned and assigned to different cores, but the dataset would also be significantly larger. We run the same query twice with cold and warm caches.

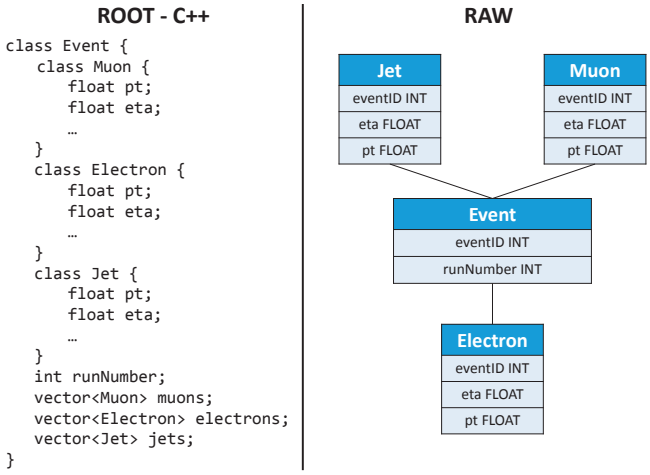


Figure 3.14 – Data representation in ROOT and RAW. The representation that RAW uses allows vectorized processing.

First Query (Cold Caches)	**Execution Time (s)**
Hand-written C++	1499 s
RAW	1431 s
Second Query (Warm Caches)	**Execution Time (s)**
Hand-written C++	52 s
RAW	0.575 s

Table 3.3 – Comparison of hand-written C++ Higgs Analysis with the RAW version.

As shown in Table 3.3, we compare the execution time of the Higgs query in RAW with that of the existing hand-written C++ code. In the first query, the execution time of RAW and of the C++ program are in the same order of magnitude. I/O is the bottleneck in both cases. RAW, however, utilizes JIT access paths to generate code similar to the hand-written C++. The important observation is that no performance is lost when querying raw data. In fact, RAW is slightly faster than the hand-written C++ due to its columnar execution model. The hand-written C++ code does not employ columnar execution; writing vectorized code by hand is difficult in practice and more so for the average user. Instead, the C++ code processes one event at a time followed by its jets/electrons/muons. This processing method also leads to increased branches in the code.

After the first query, both RAW and the hand-written C++ have populated an in-memory cache with the events of interest. In the hand-written case, this cache is ROOT's internal buffer pool, which stores previously loaded, i.e., hot, objects. In the case of RAW, the in-memory cache is built as a side effect of the column shreds strategy. Therefore, the in-memory tables of RAW are not fully populated. Instead, only attributes requested by the query exist in each table. Moreover, for a given attribute, data is only available for those rows that were actually needed during the query execution; the remaining rows that were not read - because a previous filter condition in a different attribute failed - are marked as not loaded.

In the second query, RAW is two orders of magnitude faster than the hand-written C++ code. The reason for this speedup is that all data of interest is cached in-memory in columns, which achieve better cacheline utilization and allow for vectorized operators that have code paths with fewer branches. More interesting, however, is the aggregate behavior for both queries. In the first query, RAW loses no performance even though it queries data directly from the raw files. In the second query, RAW performs as if the data had been loaded in advance, but without any added cost to actually load the data.

Discussion. The results show how adapting a query engine to the underlying raw file formats, realized using JIT access paths and column shreds, is feasible in practice and performs well in a scenario where using a relational database, which requires data loading, would be cumbersome. Besides duplicating a great amount of data in a vendor-specific format, the restrictions that relational DBMS place on a table's number of columns hinder loading data files that potentially include tens of thousands of attributes, and introduce non-trivial decisions on table partitioning. With RAW, data does not have to be loaded. In addition, analysis using RAW is faster compared to using existing hand-written algorithms.

3.7 Summary

Databases deliver impressive performance for large classes of workloads, but require data to be loaded to operate efficiently. Data loading, however, is a growing bottleneck as data volumes continue to grow exponentially and data is becoming more varied with a proliferation of new data formats. In an ideal scenario, the database query engine would seamlessly adapt itself to the data and ensure efficient query processing regardless of the input data formats.

This chapter proposes the adaptation of a query engine to the underlying data formats and incoming queries. We implement RAW, a prototype query engine manifesting this design. RAW employs a novel data access method, Just-In-Time access paths, enabling it to adapt to data formats seamlessly. JIT access paths are faster than traditional in situ query processing and competitive with DBMS for some file formats, whilst having no data loading overhead.

There are inherent overheads to raw data access even with JIT access paths, such as the cost of converting data between the raw data representation and the query engine representation. RAW uses columns shreds, a novel method that reduces these inherent overheads by pushing scan operations up in the query plan so that data is only loaded when it is strictly needed.

RAW has been successfully applied to a real-world example for which using a traditional DBMS is problematic, achieving a two-order of magnitude speedup against the existing solution, which is based on hand-written C++ code.

4 Just-in-time Query Engines

Industry and academia are continuously becoming more data-driven and data-intensive, relying on the analysis of a wide variety of heterogeneous datasets to gain insights. The different data models and formats pose a significant challenge on performing analysis over a combination of diverse datasets. Serving all queries using a single, general-purpose query engine is slow. On the other hand, using a specialized engine for each heterogeneous dataset increases complexity: queries touching a combination of datasets require an integration layer over the different engines.

This chapter presents a system design that natively supports heterogeneous data models and formats, and also minimizes query execution times. For multi-model support, the design uses an expressive query algebra which enables operations over various data models. For minimal execution times, it uses a code generation mechanism to mimic the system and storage most appropriate to answer a query fast. We validate our design by building Proteus, a query engine that natively supports queries over CSV, JSON, and relational binary data, and specializes itself to each query, dataset, and workload via code generation.

4.1 Introduction

The ongoing data explosion is leading to a major overhaul in a range of scientific and business domains. Practitioners have evolved into data scientists, relying heavily on data analysis over an increasing number of datasets. Besides relational tables, semi-structured hierarchical data formats have become the state of the art for data exchange. In addition, scientists use domain-specific formats and external structured files containing data modeled as tables, hierarchies, and/or arrays. Users execute widely different analysis tasks over all these data types. Heterogeneity, both in data and in query workload, significantly affects the way data analysis is performed.

Meaningful data analysis depends on combining information from numerous heterogeneous datasets: data-intensive domains, such as sensor data management and decision support

based on web clickstreams, involve queries over data of varying models and formats. Users that want to perform analysis over heterogeneous datasets can use a database engine that supports multiple use cases, but this approach is expensive because such engines are typically overly generic and hard to optimize for all cases. Therefore, users typically settle for a dedicated, specialized system for each of their use cases [233]. Each of these two extremes either offers i) extensive functionality and expressiveness, or ii) minimizes response times in a particular scenario, but not both. Hence, performing analysis effortlessly and efficiently remains an open problem.

One proposed solution is to flatten the different datasets into the relational model and load them in an RDBMS [225]. Data types such as hierarchies, however, are not a natural fit for tables. Another alternative is the data federation of heterogeneous data sources [65, 99]. The dominant approach in this case is packaging together multiple query engines, using the appropriate one for each specialized scenario, and relying on a middleware layer to integrate data from different sources. Thus, besides the challenge of data integration, users face a system integration issue, which increases complexity. Alternately, data analysis frameworks [43, 238] keep data in a "data lake" regardless of its format. Native support for rich data models in these systems is typically limited because it complicates system architecture and query optimization. Queries over complex data therefore incur a performance penalty. An encompassing design choice of the previous approaches is that all datasets have to be fully ingested and converted into a default format per system, either as a pre-loading step or during query answering. This process adds an additional upfront cost per query.

This chapter presents a system design that bridges the conflicting requirements for generality in analysis and minimal response times. The design supports both relational as well as nested data by using an expressive, optimizable query algebra that is richer than the relational one. The algebra allows combining data of heterogeneous models and produces data-model-conscious query plans. We couple this powerful query algebra with on-demand adaptation techniques to eliminate numerous query execution overheads. Specifically, our design is modular, with each of the modules using a code generation mechanism to customize the overall system across a different axis. First, to overcome the complexity of the broad algebra, we avoid the use of general-purpose abstract operators. Instead, we dynamically create an optimized engine implementation per query using code generation. Second, to treat all supported data formats as native storage, we customize the data access layer of the system based on the underlying data at query time. Finally, to mimic the storage that better fits the current workload, we materialize in-memory caches and treat them as an extra input. The shape of each cache is specified at query time, based on the types of data accessed and the query workload trends. Overall, the originally distinct modules collapse into a unified, specialized query engine at runtime.

We validate the proposed design by building *Proteus*, an analytical query engine that queries heterogeneous datasets without converting them to a homogeneous form. Proteus couples a general query interface with the execution times of a system that has been specialized for a specific query, data, and workload instance. Proteus currently supports CSV, JSON, and relational binary data; adding support for more formats is straightforward.

Contributions. The contributions presented in this chapter are the following:

- We present a system design principle that offers i) generality in analysis and ii) minimal response times. To achieve this, the design couples i) a query algebra that supports both relational and nested data with ii) on-demand customization mechanisms that collapse all layers of the system architecture at query time. The final result is a highly-optimized specialized engine per query.
- Based on our design, we implement Proteus, a full-fledged analytical query engine that queries CSV, JSON, and relational binary data transparently and efficiently. Proteus uses code generation to specialize its entire architecture per query and to craft caching structures of different shapes to adapt to the workload.
- We show that Proteus outperforms state-of-the-art open-source and commercial solutions in a mix of workloads. We perform a fine-grained evaluation over TPC-H data using multiple data representations; Proteus performs as if it has been designed for each use case. We also execute a challenging real-world workload over a mix of diverse datasets, in which Proteus is $\sim 3\times$ to $9\times$ faster than the state-of-the art alternatives.

Outline. The rest of this chapter is structured as follows: Section 4.2 presents related work. Section 4.3 presents the rich query algebra that Proteus uses. Section 4.4 introduces the high-level architecture of Proteus. Section 4.5 details how Proteus customizes itself on-demand to fit the requirements of each query, and Section 4.6 presents its adaptive caching capabilities. Section 4.7 experimentally validates Proteus. Finally, Section 4.8 concludes the chapter.

4.2 Related Work

A large body of work proposes a variety of solutions for the problem of querying heterogeneous data and efficient query processing in general. This section surveys related work and highlights how Proteus pushes the state-of-the-art even further.

Data Federation. To cope with data heterogeneity, data federation approaches perform analysis over diverse data sources without placing all data in a single system [30, 71, 75, 217, 239]. In recent years, the dominant approach has become bundling together multiple systems, each with a different query engine, and using the most appropriate engine for each scenario. These *polystore* systems initially combined Hadoop with an RDBMS [28, 95]. Newer proposals bundle more engines to better fit more use cases, each with a different query engine, and use the appropriate one for each specialized scenario [65, 99]. To treat multiple engines

as one, the overall solution uses middleware to perform cross-system query optimization, query splitting, data exchange between systems, etc. Thus, besides data integration, system integration becomes a concern which complicates data analysis.

To address this concern, ViDa [144] envisions effortlessly abstracting data out of its form and manipulating it regardless of its structure. This chapter describes how to realize the goals of ViDa by materializing a modular system design for queries over heterogeneous data formats. The distinct modules of the design fuse at query time, eventually resulting in a specialized implementation per query. We couple this architecture with ad hoc storage structures to adapt to the query workload.

Native Engine Support for Heterogenous Models. Commercial systems like System RX and XML DB are hybrids offering native support for both relational and XML data. System RX [53] uses XML-specific storage, an XQuery compiler, and XML indexes. XML DB [186] calibrates XML storage between CLOBs and objects "shredded" to rows. Recently, Oracle proposed extending an RDBMS with a JSON datatype [171]. SAP also discusses hierarchical data support in HANA [64], proposing language constructs, a new data type to mask the data complexity, and an indexing scheme. The processing primitives of these approaches target particular formats (e.g., relations and XML), while Proteus customizes itself for a multitude of formats on demand; its operators are by design agnostic to the underlying data for extensibility.

Encoding Schemes for Heterogenous Models. Various works advocate "shredding": flattening hierarchies and storing them in one [76, 77] or (typically) multiple relational tables (a technique called "shredding") [57, 109, 225, 66]. MonetDB [57] uses specialized data encodings, join methods, and storage for XML data. Argo [74] proposes similar encoding schemes for JSON. Shredding approaches pay a penalty to reconstruct complex objects because multiple joins are required to re-stitch an object. Finally, Sinew [236] and PostgreSQL use a custom binary serialization for JSON. Instead of fitting data to the query engine, Proteus specializes itself based on the data and query types. It operates natively over the original data instead of loading data using complex encodings. If needed, Proteus can materialize data subsets of interest into caches to emulate different encodings dynamically.

(SQL-on-)Hadoop & Cloud Systems for Heterogeneous Models. Multiple systems have been built over Hadoop or a similar distributed runtime environment to query heterogeneous datasets [39, 43, 54, 198]. Jaql [54] and Pig Latin [198] are query languages for semi-structured nested data, and both get translated to MapReduce jobs. SQL++ [199] is a recent data model and query language proposal for relational and semi-structured data, which is gradually adopted by numerous scale-out data stores. Spark SQL [43] introduces relational processing support over (semi-) structured data. Nested datatypes are again treated as objects that are opaque to the optimizer. Finally, Dremel [182] flattens nested data into columns and allows queries using an extension of SQL. This columnar representation led to the popular Parquet file format [9]. Dremel also influenced the creation Drill [5], an open-source scale-out analytical engine with Dremel-like architectural design choices.

Our work is applicable to the engines of these frameworks. For example, most of these systems use data serializers, such as Avro, to fully transform input datasets into a format they can process. Proteus, however, relies on input plug-ins that process only the data needed, and calls them at different steps of execution to judiciously convert input values, unnest nested structures, etc. Using plug-ins that are tightly integrated with the rest of the engine instead of "black boxes" that blindly ingest data can benefit these systems.

Code Generation. Runtime code generation is an established mechanism, used by several relational engines [153, 159, 190, 213]. HIQUE [159] generates cache-conscious code via templates. HyPer [190] uses the LLVM compiler [162] to generate machine code. LegoBase [153] goes through numerous rewriting ("staging") steps to generate C code. Proteus follows the HyPer paradigm and relies on LLVM too. Proteus is more expressive than relational code-generation engines because it supports multiple data models and transformations between them. Moreover, Proteus treats each supported data format as its native storage and adapts to incoming queries better because it makes dynamic decisions about its data access mechanisms, "tuple" structure, and cache organization, all of which are predefined in other systems.

4.3 An expressive query algebra

We want to enable queries over a multitude of data models, hiding the underlying heterogeneity. Thus, our query algebra must treat all supported data types as first-class objects in terms of both expressive power and optimization capabilities, instead of considering richer types as BLOB-like values which are opaque to the query optimizer. Existing approaches follow two main directions to deal with the data model variety. Each of them, however, sacrifices either generality or query performance.

The first approach involves building an entire system with a specific data model in mind and specialized to the use case at hand. A prominent example is the use of column-oriented DBMS for analytical relational workloads. Following the same trend, systems like CouchDB and MongoDB emerged for semi-structured data. Given that they are optimized for non-relational cases, they impose a number of restrictions for more "traditional", relational-like workloads. For example, data entries are assumed to be de-normalized as self-contained objects, so joins are challenging to express. Because each specialized system supports only a specific type of input efficiently, users resort to system integration, i.e., having a dedicated system for each of their dataset types and using a mediation layer over them to handle cross-dataset queries.

The second approach is to extend an established system with support for additional data types, e.g., adding support for JSON to an RDBMS. The extension is typically inefficient: A proper extension would add explicit query operators to support the new types of data, which requires significant engineering effort, as well as extending the (relational) model to which every system component adheres. Due to these constraints, commonly only functions that access and manipulate the new complex data are introduced, and the system's optimizer remains unaware of the new data type particularities.

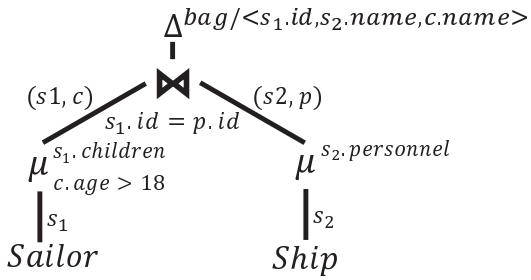


Figure 4.1 – Query involving unnest operators: Without them, the operators higher in the tree would have to process BLOBs repeatedly every time they need a nested value.

We use a third, different approach to allow queries across data of various models: We leverage a unifying data model and a powerful query language internally. Specifically, Proteus is built around the *monoid comprehension calculus* [105] because this calculus supports various data collections (e.g., bags, sets, lists, arrays) and arbitrary nestings of them. The monoid calculus and its corresponding algebra are optimizable and allow transformations across data models, hence Proteus can produce multiple types of output. The calculus is also expressive enough for other query languages to be mapped to it as syntactic sugar: For relational queries over flat data (e.g., binary and CSV files), Proteus supports SQL statements, which it desugarizes to comprehensions. For more powerful manipulations of flat data (e.g., outputting results that contain nestings) and for queries over datasets containing hierarchies and nested collections (e.g., JSON arrays), Proteus currently exposes a query comprehension syntax to the user; Example 4.1 presents a query using this syntax.

Example 4.1: Suppose we have a dataset comprising sailors and a dataset comprising ships. Each sailor has an *id* field and a *children* field which contains a list of *(name,age)* pairs for the sailor's children. Each ship entry has a *name* field and a *personnel* field, which contains a list of sailor identifiers. The query "For each Sailor, return his id, the name of the Ship on which he works, and the names of his adult children" is expressed in the calculus as follows:

```
for { s1 <- Sailor, c <- s1.children, s2 <- Ship,
      p <- s2.personnel, s1.id = p.id, c.age > 18 }
      yield bag (s1.id, s2.name, c.name)
```

As described in Section 2.2, for each incoming query, the first step is translating it to a calculus expression. The calculus expression is then rewritten to an algebraic tree of a *nested relational algebra* [105]. The resulting plan for the query of Example 4.1 is depicted in Figure 4.1. Two unnest operators deal with the nestings in the data explicitly.

Overcoming Complexity. Using a rich data model and language/algebra for queries over complex data was proposed when OODBs and XML appeared [105, 106, 240, 203]. Rich models and algebras, however, lost traction due to their complexity. The more complex an algebra is, the harder it becomes to evaluate queries efficiently: Dealing with complex data leads to complex operators, sophisticated yet inefficient storage layouts, and costly pointer chasing during query evaluation. To overcome all previous limitations, we couple a broad algebra with on-demand customization.

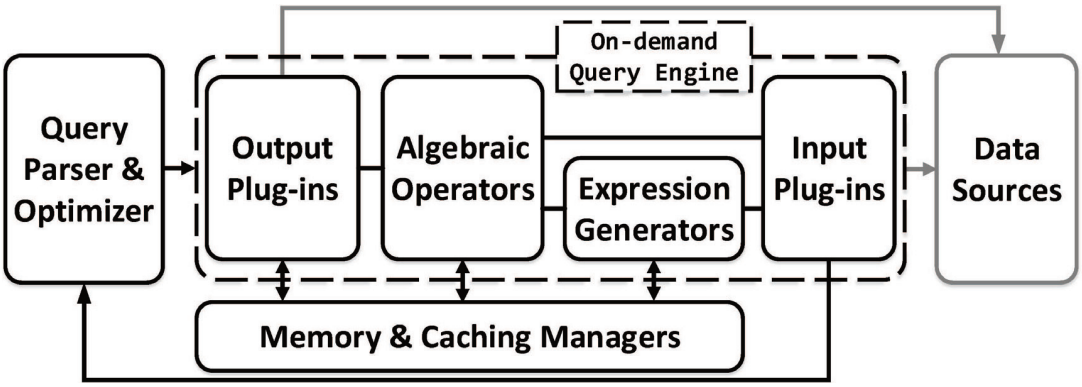


Figure 4.2 – The architecture of Proteus.

4.4 The Architecture of Proteus

Proteus is a query engine designed to enable fast queries over heterogeneous datasets. To provide generality, Proteus uses an algebra that can model operations across different types of data, thus offering expressive power and rewriting opportunities for queries targeting complex data. To also minimize response time, Proteus creates a new query engine instantiation on-demand per query via code generation. Furthermore, Proteus customizes its storage structures to adapt them to the workload. The result is a custom, highly-optimized engine, expressed in machine code and operating over a data representation that suits user analysis.

Figure 4.2 depicts the components of Proteus. The *Query Parser* handles incoming queries, which are then rewritten to a physical plan by the *Query Optimizer*. *Algebraic Operators* encapsulate data model heterogeneity; they express the plan of a query and coordinate code generation. *Expression Generators* generate code for expression evaluation when requested by an operator. *Input Plug-ins* encapsulate data format heterogeneity; they consider source-specific optimizations and generate code that accesses any required data. They also provide statistics and costing formulas per data source. *Output Plug-ins* generate code that handles operator output and cache creation along with the *Memory & Caching Managers*.

Query Optimization. Systems that process heterogeneous data face the following challenges: First, queries over hierarchical data typically involve many levels of nesting, which increases execution overheads. Second, unless an optimizer has access to data statistics, it may produce suboptimal plans. Proteus uses a three-step approach to address these issues: First, when a user asks a query, Proteus parses and normalizes it, performing operations such as selection pushdown and unnesting multiple types of nested queries. Then, Proteus rewrites the query to a nested relational algebra. The algebraic representation is amenable to relational-like optimizations and further unnesting. Finally, after a number of rule-based rewrites, the optimizer considers cost-based transformations; it follows a bottom-up strategy and relies on gathered statistics to perform access path selection and join re-ordering. Its difference from traditional optimizers is that statistics and costing of data accesses are provided by the input plug-ins relevant per query.

On-demand Query Engine. The operators of traditional query engines are hard-coded to a database-specific input data format for efficiency. Proteus is designed to treat each data format as native storage. To cope with data heterogeneity, Proteus masks data complexity from the operators by using an input plug-in per data format. Each plug-in exposes a uniform interface that the rest of the engine uses to consume data values. The algebraic operators process input either by calling expression generators or via direct interaction with an input plug-in. This separation of concerns makes Proteus extensible: adding a plug-in suffices to support a new data format.

The operators of Proteus call output plug-ins to handle the creation of output and the materialization of any required intermediate results during query execution. Proteus also uses the output plug-ins to define caching structures, which it populates as a side-effect of execution to adapt to the overall workload. Once materialized, Proteus treats caches as an additional input dataset.

For each query, Proteus uses a code generation mechanism to collapse the layered architecture of the engine – the dashed part of Figure 4.2 – into a specialized piece of code. Each of the components produces low-level machine code that Proteus combines to form a program serving the currently processed query. Specifically, once the optimizer has produced a physical plan, Proteus traverses it recursively until it finds the datasets to access (i.e., the leaf nodes). It then triggers the appropriate input plug-ins to generate code accessing data. As the recursion is returning control to the root node of the plan, Proteus generates code for every visited operator. Each visited operator may (re-)trigger input and output plug-ins to process its input and/or materialize its output.

Memory Manager. The Memory Manager handles the request of system components for memory blocks to read/write. The Manager distinguishes between input files and caching structures: It memory-maps input files, treating all input data as if it is memory-resident, and delegates paging to the OS virtual memory manager. As for caching structures, Proteus pins them in a memory pool, and uses an LRU variation to evict them when appropriate.

4.5 On-demand query engines

Ideally, a system must allow diverse queries over heterogeneous datasets, enabling cross-model and cross-format queries, but also perform as if it has been designed for a specific use case – even better, as if it is hard-coded to serve a specific query: For analytical queries over flat (e.g., binary, CSV) data, the system must be as fast as an analytical relational engine. For hierarchical data, it must be as fast as a document store.

The nested relational algebra of Proteus enables querying complex data types and considers them as first-class citizens during query optimization. It also facilitates query unnesting – a common issue when input data is nested. Dealing with complex data and query operators, however, comes at increased cost.

Even when dealing with the strictly relational operators of an RDBMS, interpreting the query plan is costly. A source of overhead is the ubiquitous Volcano iterator model [117], which enables pipelining and exposes a single interface for all operators, but complicates control flow and introduces multiple function calls per tuple processed (e.g., each operator calling *getNextTuple()*). Another factor is the variety of datatypes that each operator must be able to process: An operator must trigger different code paths depending on whether its arguments are i) integers, ii) floats, iii) some combination, etc. To support this behavior, operators use control flow statements and (virtual) function calls in their code, which leads to increased branching in the critical path of execution.

This *interpretation overhead* [159, 190], stemming from function calls and control flow statements that disrupt the instruction pipeline, affects pipelined query execution negatively. Intuitively, the nested relational algebra operators face similar issues. Even worse, they have to i) support additional, more complex types of input, and ii) perform extra work compared to their relational counterparts. For example, besides the selection and join operators, many additional operators of the nested relational algebra have an embedded filtering step (e.g., unnest, reduce). The additional complexity further increases the interpretation overhead.

One way to remove the interpretation overhead is to use a block-oriented, operator-at-a-time execution model, as columnar engines typically do [58]. The block-oriented model, however, introduces materialization overhead per operator. This cost would be more severe for Proteus compared to traditional relational systems because of the more complex datatypes to be materialized. Even worse, Proteus serves datasets whose contents rarely reside in explicit data blocks, so every query would pay an upfront cost to materialize input blocks. Instead of processing data blocks, Proteus pipelines data through its operators, but also minimizes interpretation overhead by customizing itself when it receives a query based on **i)** the query requirements and **ii)** the datasets the query touches.

4.5.1 An Engine per Query

Traditional pipelined query engines execute a query by interpreting its physical plan and invoking multiple general-purpose operators for each input tuple. Proteus removes interpretation overhead by traversing the query plan only once and generating a custom implementation of every visited operator. Proteus thus uses control flow mechanisms, such as datatype checks, only during the single traversal and avoids the per-tuple penalty that a static pipelined engine incurs. Once all plan operators have been visited, Proteus blends the generated code stubs into a hard-coded query engine implementation which is expressed in machine code.

Proteus uses LLVM [162] to generate low-level code, which it compiles at runtime. LLVM is a collection of compiler infrastructure that offers frontends for languages such as C/C++ and Fortran. In its core, LLVM translates these languages into an intermediate representation (IR) resembling assembly code: the *LLVM IR*. LLVM then compiles the IR into actual machine code based on the underlying hardware. Proteus generates LLVM IR because i) it is strictly-

typed and less error-prone than macro-based C++ code, ii) it compiles much faster than macro-based C++ code, and iii) LLVM offers rewrite passes such as dead code elimination that optimize the generated IR. In summary, Proteus uses LLVM as a plan rewriting mechanism, and performs one extra step compared to traditional query engines: It rewrites the physical algebraic plan – an abstract, high-level IR – into the imperative, low-level LLVM IR which is amenable to compiler-centric optimizations [14].

After parsing and optimizing a query, Proteus traverses the physical plan of the query in post-order depth-first-search (DFS). When visiting a node of the plan, Proteus i) visits the node's children to produce the code corresponding to their functionality, ii) generates the physical implementation corresponding to the current node, and iii) returns control to the node's parent to continue the code generation process. The recursive traversal terminates when it reaches a leaf node (a scan operator). Proteus then generates a code stub that, when executed, will launch a scan over a dataset. In each scan iteration, the generated code will access a "record" from the data and place the fields needed for the rest of the plan in virtual memory buffers. The virtual buffers can be thought of as local variables placed in the stack frame. To maximize locality, the LLVM compiler promotes buffer contents to CPU registers when possible. Therefore, subsequent operators referencing values that exist in register-backed buffers experience minimal access times and fully pipeline data. Once Proteus has generated code stubs for a leaf node, it shifts control to the node's parent, also passing along pointers to the virtual buffers and to the currently "hollow" parts of the overall query code that need to be filled in next. The same process continues until control returns to the root node.

Figure 4.3 depicts a plan for the query `SELECT COUNT(*) FROM A WHERE e`, along with a high-level description of the resulting code. The scan of relation A results in the generation of a "hollow" while-loop. The code for the ending condition of the loop (line 1), as well as for populating virtual buffers with the fields necessary to answer the query (line 2), is injected by an *input plug-in* that allows the data-format-agnostic scan operator to interface with dataset A regardless of how it is stored. Then, the selection operator generates a hollow *if* block, whose outcome depends on the evaluation of the expression *e* (line 3) in each iteration. Proteus retrieves the values required to evaluate *e* from the virtual buffers. The reduce operator calculates the final result by incrementing a counter, which it then outputs. The result of the physical plan traversal is not a number of standalone operator implementations: It is a minimal, specialized piece of code representing an entire query, with operator logic tightly stitched together to ensure pipelined query execution. This type of execution minimizes intermediate query results, maximizes code and data locality, and reduces register pressure.

Proteus also uses pre-existing (i.e., not generated) C++ code for some of its functionality. Proteus wraps these operations in C++ functions and calls them when appropriate from the generated code. For example, the Memory and Caching Managers do not generate code. In another case, Proteus uses hash-based algorithms for the join and grouping operators, namely variations of the radix hash join algorithm [174] adapted from [46]. While parts of the join implementation are indeed generated at runtime, other parts, like clustering the materialized

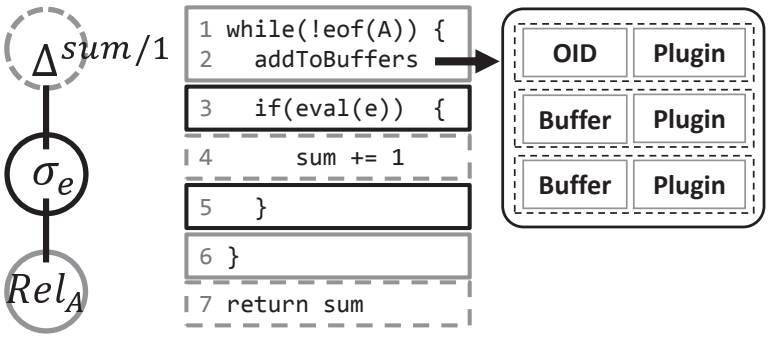


Figure 4.3 – Example of a query plan and of the generated (pseudo-) code. Once the scan operator places needed fields in virtual buffers, they are used to evaluate the filtering expression.

entries based on their hash values, are wrapped in a C++ function. This function is only called once per join side, so the overhead of making the function call is minimal.

Implementation. The layers of Proteus that parse, rewrite, and optimize queries are expressed in Scala and output a physical query plan. The layers that traverse the query plan and trigger code generation using LLVM are written in C++. When Proteus receives a query, it generates stubs of LLVM IR, which it stitches together during the traversal of the physical query plan and puts into a single function. Within milliseconds, LLVM compiles the IR of the function into actual machine code based on the underlying hardware. The result is a library, which Proteus calls to serve the query.

4.5.2 A Custom Data Access Layer per Query

The operators of Proteus access a dataset either by triggering an expression generator to produce code for the evaluation of an algebraic expression, or by directly calling the corresponding input plug-in. This separation of concerns ensures extensibility.

Expression Generation

Proteus places values from each dataset it touches into virtual memory buffers, which the query operators use to evaluate expressions of the nested relational algebra. For example, if Proteus has populated buffers with fields $a.sal$ and $a.bonus$, it can evaluate the filtering expression of the operator $\sigma_{sal+bonus<3000}$. The physical operators assign the evaluation of algebraic expressions to an expression generator. In the example of Figure 4.3, an expression generator produces the code to calculate the result of $eval(e)$ at line 3, and injects it as the condition in the *if* statement. Similar generators are used when hashing an expression and when flushing out the query output. A useful property of this separation is that the operators are agnostic to the underlying data models/formats/properties. The operators are oblivious to whether a value in the memory buffers belongs to an array, is nested, or is not fully materialized yet; all they require is that the expression generators inject the appropriate code for expression evaluation at the code spots they designate.

Input Plug-in Methods		
generate()	hashValue()	unnestInit()
readValue()	flushValue()	unnestHasNext()
readPath()		unnestGetNext()

Table 4.1 – The input plug-in API of Proteus.

Input Plug-ins

Proteus masks the details of the underlying data values from the query operators and the expression generators. To interpret data values and generate code evaluating algebraic expressions, Proteus uses *input plug-ins*. Each input plug-in is responsible for generating data access primitives for a specific file format. Proteus currently uses input plug-ins for CSV, JSON, and relational binary data (both row-oriented or column-oriented).

Table 4.1 lists the API that every input plug-in exposes. Calls to a plug-in can be made by i) a scan operator populating virtual memory buffers (the *generate()* call), ii) an unnest operator looping through a nested collection (*unnestInit()* etc.), or iii) an expression generator producing the code to calculate the result of an expression. In the third case, *readValue()* provides a field's value to the expression generator, and *readPath()* returns a pointer to a data object's field. Consecutive calls to *readPath()* are used to access nested fields.

When a scan operator calls an input plug-in, the plug-in generates code that customizes the data access layer of Proteus based on **i)** the *current query requirements* and **ii)** the characteristics of the dataset to be accessed: its *schema*, *format*, and *contents*. Using this information, Proteus generates code that performs fewer and more efficient data accesses than a general-purpose scan operator. An example of exploiting the query requirements is the following: During query rewriting, Proteus pushes field projections down to the scan operators so that it extracts only the fields necessary. To perform these selective accesses, a general-purpose scan operator would use a loop that checks whether each field is needed for the query, thus introducing branches in the critical path of execution. Instead, Proteus generates code processing only the required data fields. Proteus also uses the dataset schema to avoid unnecessary control logic, such as datatype checks – it generates specific access primitives for integer fields, nested fields, etc. The overall code generated for scanning data resembles a hard-coded program.

Specializing per Dataset Format. Proteus generates code that considers the particularities of each data format. For binary relational data, an input plug-in generates code reading the memory positions of the required data fields. For more verbose or richer formats, Proteus uses more sophisticated access methods. The common denominator of all input plug-ins is that for every data object / "tuple" they access, they produce an object identifier (*OID*), which they forward to the query operators. As an example, for flat data the OID is a row counter. Using an entry's OID, an expression generator can invoke the corresponding input plug-in at a later point in execution to access a value needed for an expression's evaluation.

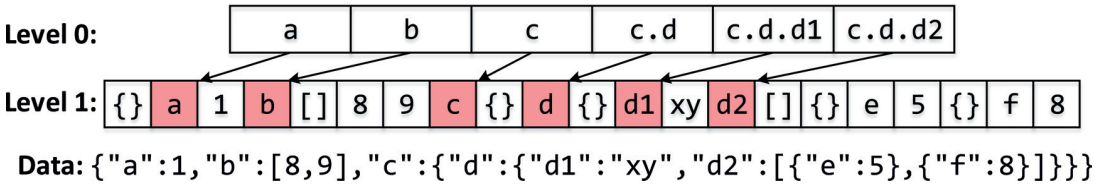


Figure 4.4 – Example of a structural index for a JSON object.

Apart from creating an OID, Proteus calibrates how eager / lazy the generated access primitives are (i.e., which values to place in memory buffers apart from the OID, whether to eagerly convert a value to a binary serialization, etc.). Proteus supports lazy plug-in behavior because eagerly populating memory buffers may prove unnecessarily expensive. When performing a path query over nested objects or data unnesting, Proteus avoids eagerly serializing a complex object only to process a subset of it: Instead, Proteus uses structural information for the data to navigate in the dataset and to access only the values necessary to provide a result. In addition, in many cases Proteus delays data conversion because it may prove to be unnecessary (e.g., because of some selection filtering out results). Another scenario is applying different materialization policies in relational workloads. To allow for this flexible behavior and enable a field's reconstruction at any point, Proteus maintains plug-in information for each field value in its memory buffers. For every such value, the corresponding plug-in uses rules to specify how lazily to process it based on criteria such as its data type and at which point of the query it is used, and generates appropriate code.

Structural Indexes. The input plug-ins of Proteus use auxiliary structures to reduce the navigation cost associated with verbose data formats, for which every access requires substantial parsing effort. These *structural indexes* store *positional* information about fields in the datasets instead of actual data values. Their entries are addressable by OID, so that all plug-ins have uniform behavior.

For CSV datasets, structural indexes store the binary positions of a number of data columns in each row [36]. Proteus stores the position of every Nth field of the file (e.g., if N=10, it stores the positions of the 1st, 11th, ... fields). When looking for a field, Proteus locates the closest indexed field position and starts seeking from that point.

Structural indexes for JSON require a more involved process because of the inherent complexity of the JSON format, which allows arbitrary levels of nesting and field order. In addition, some (optional) fields may be present in a subset of a JSON document's objects. Overall, the semi-structured nature of JSON files complicates their validation and processing. When Proteus accesses a JSON file for the first time, it validates the input. During validation, Proteus populates an index per JSON object with structural information. The resulting structural index serves two goals: It reduces the parsing effort for subsequent accesses to the file, and minimizes the interpretation overhead stemming from the schema flexibility of JSON data.

Each entry of a JSON structural index captures information about a *token* (e.g., a field name, an array, etc.) contained in a JSON object: its binary *starting and ending positions* in the file, as well as its *type*. To serve requests for a data field, the JSON input plug-in finds its corresponding token entry in the structural index instead of re-parsing the file from scratch. The plug-in then forwards the entry identifier – which acts as an OID – to the operator / expression generator that requested it. Then, the OID is either dereferenced to access and convert the JSON value, or kept as is for subsequent processing – a case of lazy evaluation.

The structural index described so far corresponds to "Level 1" of the example in Figure 4.4. The first index entry, labeled "{}", keeps the starting and ending positions of the overall JSON object, the second entry keeps the positions of token a, and so on. Intuitively, if a dataset contained multiple objects similar to the one depicted and a query requested field a from each one, the JSON plug-in would follow the same process for each object: Return the second entry of the object's corresponding structural index. Nevertheless, there is no guarantee that field a comes before field b, b before c, etc. in every object of the dataset. Thus, Proteus would have to sequentially scan each object's index and compare the label of the wanted field with the one currently visited.

Proteus removes this overhead which stems from JSON schema flexibility by introducing an additional "Level 0" to the structural index. "Level 0" comprises an associative array which maps field names to their corresponding positions in "Level 1" of the index. The shaded values in Level 1 are now redundant and thus removed. Proteus performs dictionary encoding over the input field names: Each name is assigned a number representing its position in Level 0. The JSON plug-in therefore finds a field's position by first calculating its encoded value, and then performing a lookup to the associative array. The use of Level 0 reduces data access costs and offers determinism despite the semi-structured JSON nature.

Proteus also registers nested records in Level 0. In Figure 4.4, by storing pointers to field $c.d.d1$, dereferencing occurs in one step instead of multiple ones. Nested collections are treated otherwise: Notice that fields e and f, which correspond to the contents of a nested (array) collection, are omitted from Level 0. JSON structural indexes opt against maintaining pointers to array contents because Proteus has an explicit *Unnest* operator to handle nested collections. The code path of Unnest applies the same action to every nested element, therefore it is unaffected by schema flexibility.

Specializing per Dataset Contents. The more information Proteus obtains about a dataset, the more aggressive optimizations it performs. Given its code generation capabilities, an input plug-in can craft an optimized code path suitable only for a specific file instance. In the case of JSON data, schema flexibility introduces overhead even when using a structural index, because Proteus has to store more bookkeeping information per indexed entry, and generate more complex code to process it. There are many scenarios, however, such as the case of machine-generated data, where every object in a dataset has the same fields in the same order. Proteus can verify whether this case holds while creating a structural index, and

drop Level 0 because the lookup process is now more deterministic: It is sufficient to maintain the sizes of any variable-length structures (i.e., JSON arrays) met and combine them with fixed schema information to deterministically compute the exact positions of relevant structural index entries. The result is a more compressed structural index and an efficient code path for lookups to the JSON dataset. In a similar optimization, if a CSV file contains fixed-length entries, Proteus deterministically computes every position and injects it in the generated code instead of using a structural index.

Enabling Cost-based Optimizations. Proteus uses a metadata store to maintain statistics per data source, namely dataset cardinalities and min/max values per attribute, and delegates statistics collection to each input plug-in. The statistics collection process is fine-tuned to avoid introducing execution overheads. Specifically, Proteus refrains from generating code for statistics gathering in every query to avoid bloating the minimal generated code. Instead, it collects statistics in three ways: First, Proteus collects statistics during the first (cold) access to a dataset, because I/O masks the overhead of statistics gathering. Second, when the plan contains a blocking operator (e.g., a join), the relevant input plug-in injects code that profiles the materialized values all at once. Finally, a daemon process periodically triggers statistics-gathering queries when the system is idle – a methodology followed by multiple DBMS. Regarding costing, each input plug-in uses different cost formulas, which it instantiates with data statistics to provide cost estimates to the query optimizer. Delegating source-specific work to a "wrapper" per source is also popular with federated systems [216, 217].

Proteus allows plug-in developers to calibrate statistics gathering and costing. The baseline option is to use predefined, hard-coded estimates in place of statistics-based computations (e.g., assume that the default selectivity of a predicate is 10%), as well as textbook cost formulas. Proteus offers such a skeleton for every input plug-in by default because it has been shown to have satisfying results [216]. Regarding statistics, Proteus allows developers to adjust/change the sampling function to be called during cold queries and result materialization. The function specifies the type of sampling to be used, and on which fields the statistics-gathering mechanism should focus on. Regarding costing, the developer can change the provided cost formulas with more suitable ones for her needs.

Adding More Inputs. Adding support for more inputs is straightforward. For each new input, what is required is to code in an input plug-in which implements the methods of Table 4.1. A developer can use plain C++ instead of the lower-level LLVM API, since Proteus can directly call C++ functions, or even call sophisticated libraries for JSON parsing [11, 12, 19, 168]. The plug-in developers decide how to calibrate ease of development and high performance based on their requirements. The same trade-off applies when integrating Proteus with existing data stores such as an RDBMS: A plug-in can either i) issue SQL queries to the DBMS, or ii) directly access the proprietary binary data format that the DBMS uses internally.

4.6 Adapting storage to workload

Proteus dynamically populates data caches as a side-effect of query execution to adapt to the workload trends. These caches can be viewed as dynamic materialized views [157], following the data recycling principle [136, 189] of automatically caching results during query evaluation for possible reuse in the future. Proteus deals with complex models and formats, so the importance of reuse is even higher because of the effort needed to re-access the data involved and recompute the expressions that queries require. Since users express a range of queries over a variety of data, the caches must facilitate each diverse workload, adapting to serve it efficiently. Therefore, instead of having a predefined structure, the caches adapt to the types of queries asked. Depending on the query workload, the caching structures can resemble i) pages filled with tuples in a system's buffer pool, ii) binary columns accessed by a columnar engine, iii) nested objects serialized in a binary format, etc.

Proteus can cache any expression supported by the nested relational algebra. Each query may trigger the population of caches of different shapes – caches of different shapes can even be built at different phases of the same query. For example, a query sub-tree processing hierarchical data may benefit from a different cache type than the query part touching relational tables. Some expression types that Proteus can cache are the following:

- Field projections ($rel.attrA$).
- Arithmetic expressions ($(rel1.salary + rel2.bonus) * 12$).
- New record constructions ($< rel.attrA, tree.attrB.attrB1 >$).

Proteus uses caching primarily to benefit queries over non-binary, verbose sources such as CSV. By caching data entries in a more compact binary format, neither parsing nor data conversions are required to access them. Caching is also beneficial when a different data layout is more suitable for the workload than the one currently used by a dataset [120]. Proteus is flexible enough to allow different caching policies depending on the expected workload type.

Implementation. The algebraic operators are oblivious to which expressions are to be cached and which of the input values they process is actually served from caches. When Proteus has to materialize data (e.g., during a join), or the physical plan contains a caching operator, Proteus assigns the task to an *output plug-in* that specifies i) the expression to be cached, ii) what the serialization format will be, and iii) the "degree of eagerness" to be used during caching. For example, when dealing with variable-length string entries, it might be sufficient to cache their binary starting positions, or even the OID of the entry to which they belong. Different types of workloads benefit from different policies across these axes.

Output plug-ins trigger cache construction similarly to expression evaluation: For each data entry, an expression generator produces code which evaluates the expression to be cached and places the result in a consecutive memory block. Proteus exposes the data cache as an additional input. As with the rest of the datasets, Proteus accesses the cached data using a dedicated input plug-in.

Building Caches. Proteus triggers cache creation i) *implicitly*, as a by-product of an operator's work, or ii) *explicitly*, by introducing caching operators in the query plan. *Implicit* caching exploits that some Proteus operators materialize their inputs: nest and join are blocking and do not pipeline data. Especially for joins, Proteus uses a radix hash-join variation, which materializes both input sides. It is thus important to re-use populated data structures and avoid re-building them, especially if the data originated in a verbose data format for which accesses are expensive.

For *explicit* caching, Proteus can place buffering operators at any point in the query plan. An explicit caching operator calls an output plug-in to populate a memory block with data. Then, it passes control to its parent operator. Creating a cache adds an overhead to the current query, but it can also benefit the overall query workload: When accessing verbose data formats like JSON, it is advisable to avoid re-accessing the original data whenever possible. Even when using auxiliary structures to navigate in the file, there are still additional costs. After locating a required field, the input plug-in typically needs to convert it to a binary form. In addition, in the case of JSON, verbose objects pollute CPU caches with unneeded information. Each field that Proteus needs is located at an arbitrary position in the file. Every time Proteus places it in a CPU cache line, the rest of the line is typically filled with an unneeded part of the overall JSON object. Dealing with compact, packed binary caches greatly improves data locality. Therefore, if a cached field ends up being re-used, the benefit from avoiding these data accesses and computations is significant.

Cache Matching. For every cache that Proteus populates, the Caching Manager stores the physical plan corresponding to the cache and uses it as a search key during cache matching. Proteus considers the available caches right before generating code. The cache/view matching process resembles that of [189, 218]. Proteus treats the physical plan as a DAG, where each node corresponds to a physical operator, and traverses it in bottom-up fashion. The Caching Manager traverses each stored plan simultaneously with the traversal of the query plan currently examined. For every node of the DAG visited, Proteus probes the Caching Manager for nodes in the cached plans that can be used instead. For a node in the current query to *fully match* a node in a cached plan, i) they must both perform the same operation (e.g., selection), ii) have the same arguments (i.e., evaluate the same algebraic expressions), and iii) their children nodes must match each other respectively. Whenever the Manager finds a match, Proteus applies the same process recursively until it reaches the root of a cached plan. If successful, Proteus rewrites the plan to use the cache. Besides full matches, Proteus considers *partial matching*. Specifically, if Proteus has implicitly cached the intermediate results (i.e., the hash tables) of $A \bowtie B$, then the newly arrived query $A \bowtie C$ can re-use the hashtable built for A if it uses the same join key. Future work includes adding support for *subsumption* [107, 218], i.e, identifying that the cached tree $\sigma_{x>0}(A)$ can replace the current sub-tree $\sigma_{x>10}(A)$ as long as we re-apply the $x > 10$ predicate.

In summary, rewriting scenarios include replacing i) a sub-tree of the plan (e.g., a scan and a subsequent unnest operator), ii) a single operator (e.g., a scan), or iii) a part of an operator

(e.g., one of the already materialized sides of a radix hash join). Code generation is an enabler for such rewrites of varying granularity because it allows Proteus to generate code only for the necessary operations.

Cache Policies. Selecting which views to materialize is a well-studied research problem [125]. Proteus applies different materialization policies depending on the workload characteristics. Proteus benefits significantly when it places caching operators close to the leaf nodes of the plan in order to convert input (raw) values to a binary format. A reason is that raw data access is a major overhead when querying heterogeneous datasets. In addition, the simpler an operator tree corresponding to a materialized result is, the more upcoming queries will be able to re-use it and benefit from it. Therefore, the Caching Manager currently focuses on ways to fully replace a costly access path instead of materializing the result of a complex query sub-tree; applying more sophisticated policies and studying their effect [32, 125] is part of our future work. Proteus thus opts for straightforward first-come-first-served caching policies and eagerly caches values read from CSV and JSON files. Proteus caches primitive values found in files containing hierarchies to avoid re-navigating through them, especially if the involved objects are deeply nested. Proteus also caches fields used as filtering predicates. On the contrary, Proteus avoids caching variable-length string fields from CSV and JSON files, which can be verbose and pollute the caches. Regarding cache eviction, Proteus uses a data-format-biased version of LRU, favoring data from inputs that are more costly to access (where $JSON \gg CSV \gg Binary$).

4.7 Experimental Evaluation

We evaluate Proteus using i) synthetic benchmarks to isolate the performance of common query operations, and ii) a real-life spam email analysis workload provided by Symantec.

Experimental Setup. We compare Proteus against a) systems that at some point were extended to support richer data models, and b) systems specialized for a specific scenario by design. Specifically, we compare i) PostgreSQL 9.4.1, ii) commercial DBMS X, iii) MonetDB 11.19.9, iv) commercial DBMS C, and v) MongoDB 3.0.3. PostgreSQL and DBMS X are row stores that support both relational and JSON data; they showcase how a generic system performs in the two diverse cases. We configure DBMS X to use its "main memory accelerator", which keeps data in memory using a custom memory-friendly layout. MonetDB and DBMS C are read-optimized column stores, designed to efficiently support relational analytical queries, which recently added JSON support. Finally, MongoDB is a specialized system for JSON data, for which it uses a binary serialization (BSON). PostgreSQL supports both a binary (jsonb) and a character-based JSON serialization; we use jsonb because of its efficiency. The other systems treat JSON as a subtype of VARCHAR. Neither the systems we compare against nor Proteus make assumptions about field order in the JSON files.

We run all experiments on a dual socket Xeon Haswell CPU E5-2650L (12 cores per socket @ 1.80 GHz), equipped with 64 KB L1 cache and 256 KB L2 cache per core, 30 MB L3 cache

shared, 256 GB RAM, and 2TB 7200 RPM SATA 3 disk storage. The operating system is Red Hat Enterprise Linux 7.1. Proteus uses LLVM 3.4 to generate custom code with the compilation time being at most ~ 50 ms per query. We run all systems in single-threaded mode.

4.7.1 Specializing the Query Engine on Demand

This experiment isolates the performance of typical query operations over both hierarchies and relations. We use JSON and relational binary data, and examine a range of query templates with 10%, 20%, 50%, and 100% selectivity.

We use the TPC-H **lineitem** and **order** tables as input, using scale factors 10 (**SF10** - 60M lineitem tuples, 15M order tuples) and 100 (**SF100** - 600M lineitem tuples, 150M order tuples). We shuffle each file's contents to avoid potential optimizations that exploit interesting orders and can introduce noise to the experiments. To test performance over JSON data, we convert the TPCH-SF10 tables into a 20GB JSON file for lineitems and a 3.5GB file for orders, and load them in all the systems we compare against. As an indication of storage size, PostgreSQL requires 27GB to store the JSON version of lineitem, and MongoDB requires 30GB. Proteus natively operates over the JSON files and builds a structural index during the first data access. Index size is ~ 21% of the JSON file for lineitems and ~ 15% for orders, and its construction is significantly faster than loading the data in the other systems (e.g., ~ 4× faster than MongoDB). For experiments over binary data, we load the TPCH-SF100 version in PostgreSQL, DBMS X, MonetDB, and DBMS C. Proteus operates over binary column files similar to the ones of MonetDB. All systems operate over warm OS caches. Unless otherwise specified, the adaptive caching of Proteus is deactivated. The data types are numeric fields (integers and floats).

Projections. For queries projecting a varying number of fields, we use three variations of the following query template:

```
SELECT AGG(val1), ..., AGG(valN)
FROM  lineitem
WHERE l_orderkey < [X]
```

The first two variations compute COUNT and MAX respectively. The third variation computes four aggregations (COUNT and MAX). Figure 4.5 plots results for the JSON version (SF10). Proteus is the fastest system because its lightweight generated code path makes it more efficient for the CPU-intensive task of processing JSON entries. In addition, contrarily to PostgreSQL, Proteus does not treat JSON objects as bulky BLOB data; it uses the structural index to retrieve the information it needs from each object, which it then feeds in the query pipeline without "polluting" the CPU caches with the verbose JSON object any further. As for the other systems, JSON access is expensive for DBMS X because it uses a character-based encoding. MongoDB is competitive with PostgreSQL only for the COUNT query. As the number of aggregates to compute increases, PostgreSQL outperforms MongoDB. JSON support

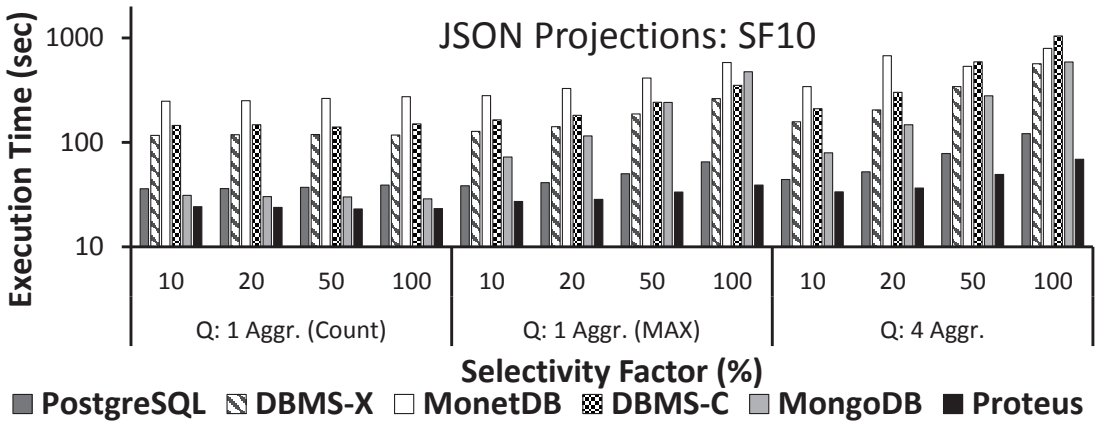


Figure 4.5 – Projection-intensive queries over JSON data.

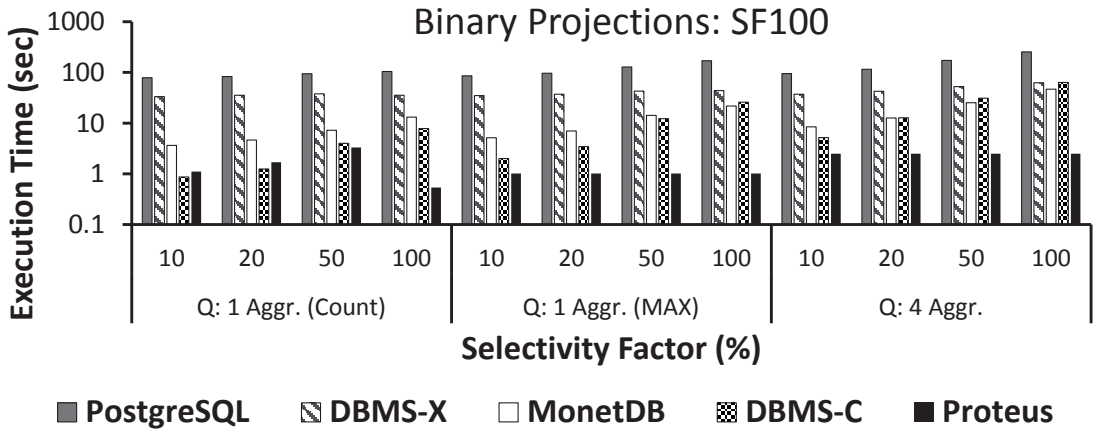


Figure 4.6 – Projection-intensive queries over binary relational data.

is still immature in MonetDB, which results in suboptimal performance. Similarly, DBMS C underperforms in all our experiments over JSON data. For this reason, and because some of the benchmarked operators are either work-in-progress (e.g., unnest) or not yet supported efficiently (e.g., using a JSON field in a GROUP BY clause requires a costly workaround for MonetDB), we exclude MonetDB and DBMS C from the other experiments with JSON data.

Figure 4.6 presents results for the queries over binary data (SF100). MonetDB and DBMS C are faster than PostgreSQL and DBMS X because the analytical query template we study is suitable for column-oriented engines (i.e., a small subset of the relation is accessed). For selective COUNT queries, DBMS C is the fastest system because it sorts the input during data loading; given that the query has a predicate on the sorting key, DBMS C exploits it to skip many data entries while answering the query. In addition, this query does not project any attributes, therefore DBMS C does not incur any tuple reconstruction cost. For less selective instances of the COUNT query and for the other more complex queries, Proteus is faster than DBMS C and MonetDB; their columnar operators produce intermediate results (i.e., fully materialize their

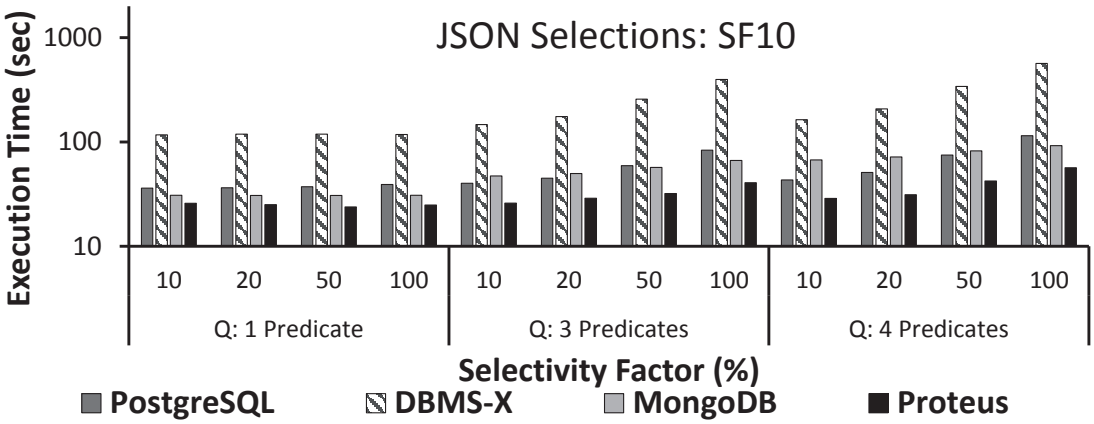


Figure 4.7 – Selection queries over JSON data.

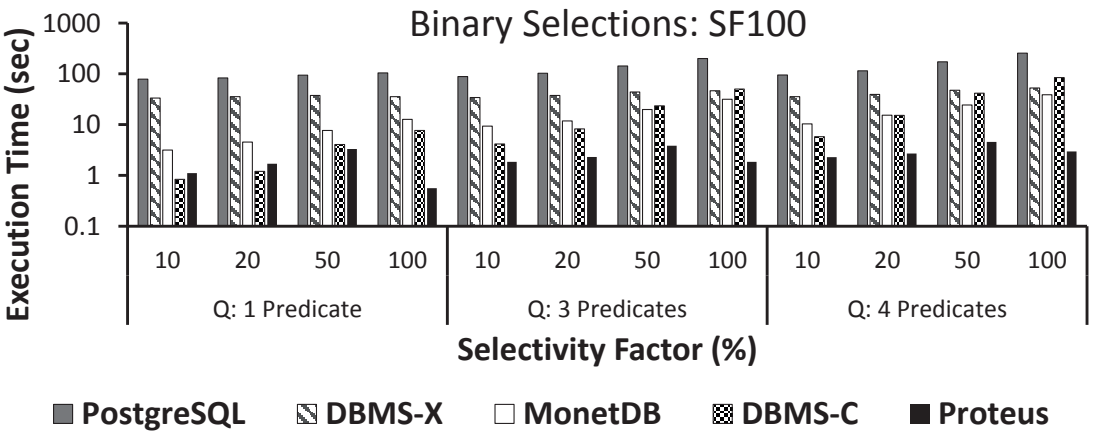


Figure 4.8 – Selection queries over binary relational data.

output), thus paying a materialization cost for the columns involved. The materialization cost increases further as queries become less selective; Proteus pipelines data instead. In addition, the resulting code of Proteus is a tight, minimal while-loop which only contains an if block evaluating the selection condition. The importance of generating minimal code is highlighted in the COUNT query (left side of Figure 4.6). The code is minimal enough for the effect of the branch predictor to be visible. When selectivity reaches 100%, very few mispredictions occur, therefore the query becomes faster for Proteus, although intuitively Proteus does more work to calculate the aggregate value.

Selections. To test queries with multiple selection predicates, we use three variations of

```
SELECT COUNT(*) FROM lineitem
WHERE val1 < [X] AND ... AND valN < [Z]
```

The queries include one, three, and four predicates in the WHERE clause respectively. Figure 4.7 presents the results over JSON data (SF10). Proteus has to convert the values it needs

on the fly, whereas PostgreSQL and MongoDB operate over a binary serialization. Still, Proteus is faster than the other systems across the whole experiment because once it has extracted the values it needs, it reduces the rest of the CPU overheads significantly. Besides pipelining, Proteus consults its structural index to pinpoint needed fields, thus reducing navigational cost in the file. These benefits become more apparent for less selective queries. DBMS X is the slowest system because of its character-based JSON encoding. Compared to Figure 4.5, MongoDB closes the gap on PostgreSQL and Proteus because the current query template projects out a count instead of more complex aggregates which MongoDB does not compute as efficiently.

In the case of binary data presented in Figure 4.8, the outcome is similar to the one for projection queries. Proteus is faster in the majority of cases because it pipelines data through all operators. MonetDB and DBMS C operators materialize their output, which becomes more expensive as selectivity moves towards 100%.

Joins & Unnests. To test joins, we use three variations of the following template:

```
SELECT AGG(o.val1),...,AGG(o.valN)
WHERE val1 < [X] AND ... AND valN < [Z]
FROM orders o
JOIN lineitem l ON (o_orderkey = l_orderkey)
WHERE l_orderkey < [X]
```

The first two variations compute one aggregation, COUNT and MAX respectively, while the third variation computes two aggregations (COUNT and MAX).

Document stores such as MongoDB do not offer first-class support for join operations, under the assumption that JSON data is typically denormalized (i.e., any joins are pre-materialized). We therefore include one more variation of a COUNT query over denormalized JSON data; each order object now contains an array with the lineitems that correspond to it, so the query has to *unnest* these JSON arrays instead of performing a join.

Figure 4.9 plots the results for the JSON case. Proteus is faster than the other systems because of i) its minimal generated code, ii) its lightweight JSON access path, and iii) the efficiency of the radix hash join algorithm it uses, which explains the larger performance gap from PostgreSQL compared to the previous query types. For MongoDB, we implement the join logic in a map-reduce-like query. MongoDB is unsuitable for such operations, which explains its poor performance; we only list its results for the first query as an indication. On the other hand, in the "Unnest" case, MongoDB outperforms PostgreSQL and DBMS X, which rely on built-in functions to perform data unnesting instead of an explicit query operator. Proteus is faster because its generated code involves almost no data conversions; besides evaluating a predicate, the code only increments a counter for each element of the nested *lineitem* arrays.

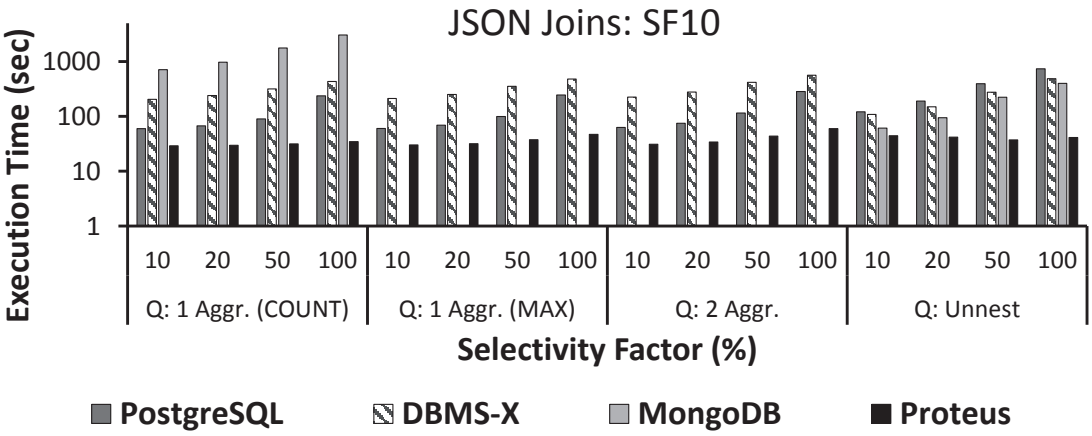


Figure 4.9 – Join and unnest queries over JSON data.

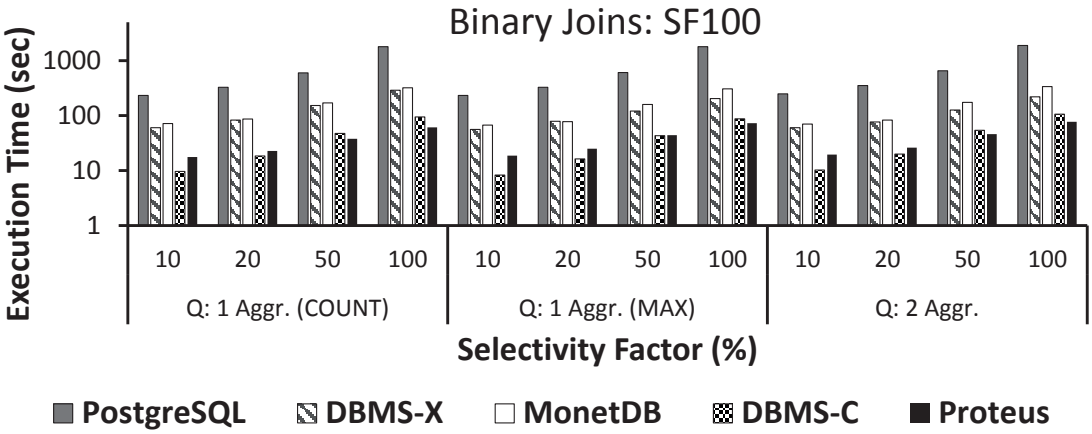


Figure 4.10 – Join and unnest queries over binary relational data.

For joins over binary data, the query template is ideal for DBMS C and DBMS X. As seen in Figure 4.10, DBMS C is the fastest system for selective queries because it exploits the fact that it sorts the data on the filtering key at loading time and thus skips multiple entries. In addition, it performs sideways information passing: it applies the filter on $orderkey$ to both sides of the join, thus reducing the pairs to be joined. DBMS X also performs sideways information passing, thus closing the gap with the column stores and Proteus, compared to previous queries. For less selective queries, Proteus is the fastest system because DBMS X and DBMS C prune fewer tuples. To further study performance, we measure performance counter statistics for MonetDB and Proteus because they use the same query plan without the additional optimizations. For a join with 20% selectivity, Proteus had 40× fewer dTLB (data Translation Lookaside Buffer) misses, 10× fewer last-level-cache (LLC) misses, and 2× fewer branches encountered, leading to fewer branch mispredictions. These factors contribute to faster response times for Proteus.

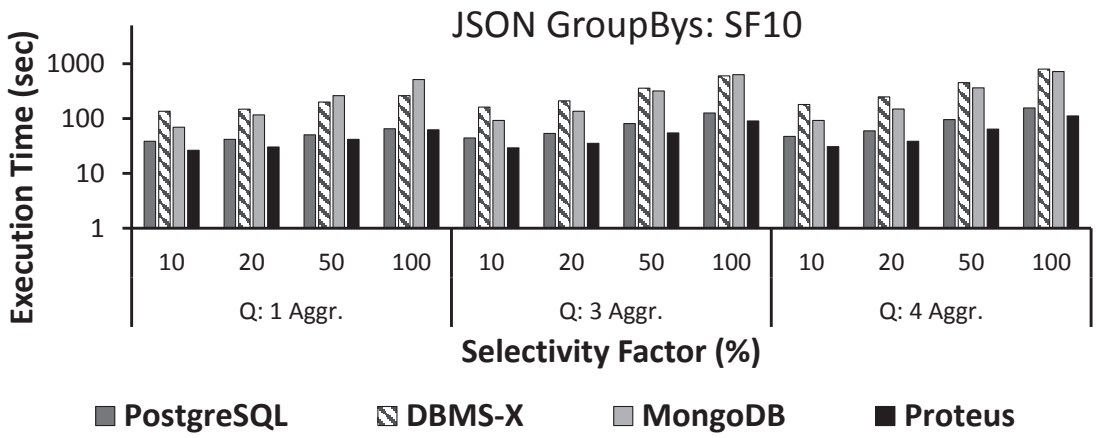


Figure 4.11 – Aggregate queries over JSON data.

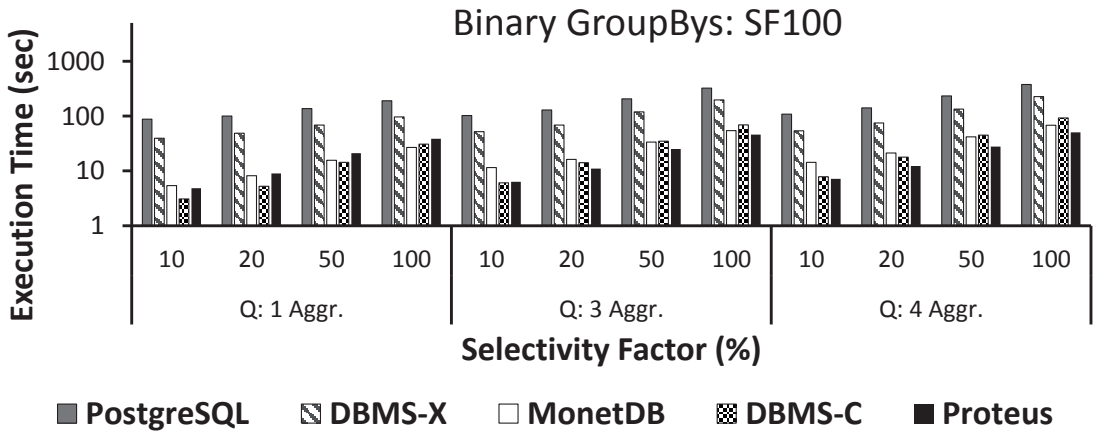


Figure 4.12 – Aggregate queries over binary relational data.

Aggregations. To test queries that group results, we use three variations of the following template:

```
SELECT AGG(val1), ..., AGG(valN)
FROM lineitem WHERE l_orderkey < [X]
GROUP BY l_linenumber
```

Figures 4.11 and 4.12 present results for queries calculating one, three, and four aggregate values. Proteus uses a radix-hash-based grouping implementation, so the results for JSON data (SF10) are similar to the join use case, with Proteus outperforming the rest. For the first query over binary data (SF100), MonetDB exploits an optimization to perform the grouping without explicitly calculating a count: It calculates the count by returning the size of each corresponding bucket in the hashtable it populates to perform the grouping. Therefore, it gradually becomes faster than Proteus when only a count is computed. DBMS C also has a headstart because it skips data based on the *orderkey* value. For queries with additional aggregates, Proteus is the fastest system.

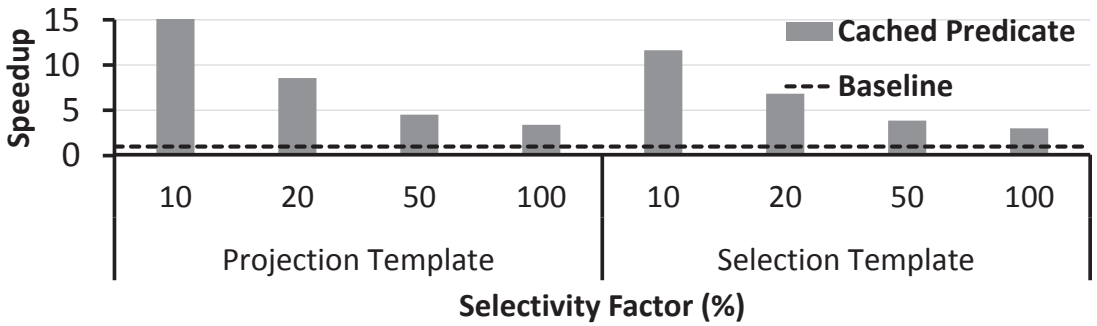


Figure 4.13 – Effect of caching on i) a projection query and on b) a selection query over JSON data.

Gauging the Effect of Caches. In the previous experiments, the caching feature of Proteus was deactivated. To quantify the speedup that Proteus can achieve by enabling caching, we instantiate the previous "projection" and "selection" templates for JSON and vary selectivity from 10% to 100%. Figure 4.13 plots the results. The first query applies a selection predicate and projects four fields. The "Baseline" dotted line is the Proteus configuration used in the previous experiments. In its "Cached Predicate" variation, the values used in the query's selection predicate are already cached by a previous query. The second query evaluates four predicates and then calculates a count. Its "Cached Predicate" version reads the values to evaluate the most selective predicate from the caches. In both queries, cache size is ~ 1.2% of the JSON file.

For the projection template, caching JSON values brings a high benefit. By touching the JSON file only to access the qualifying values to be projected, Proteus achieves a speed-up of up to 15× for selective queries. As selectivity reaches 100%, Proteus avoids fewer accesses of the JSON file, therefore the speedup is lower. We observe significant speedup for the selection template as well. The speedup is smaller than in the case of the projection query, because even though the projection-intensive query is more expensive than the selection-intensive one in its baseline version, both of them end up having the same execution time under "Cached Predicate". In other words, there are some constant costs (e.g., structural index navigation) which define the minimum execution time.

Summary. Proteus is competitive with specialized systems for different operations regardless of the underlying data models and formats. We also saw the additional benefits brought by caching, which we investigate further in the next section.

4.7.2 Adapting to a Real-world Workload

We now evaluate Proteus using a workload obtained from Symantec, which performs analysis over data derived from spam e-mails. The data silo of Symantec periodically receives batches of JSON files, collected through worldwide-distributed spam traps. Each file contains information about spam e-mails, such as the mail body and its language, its origin (IP address,

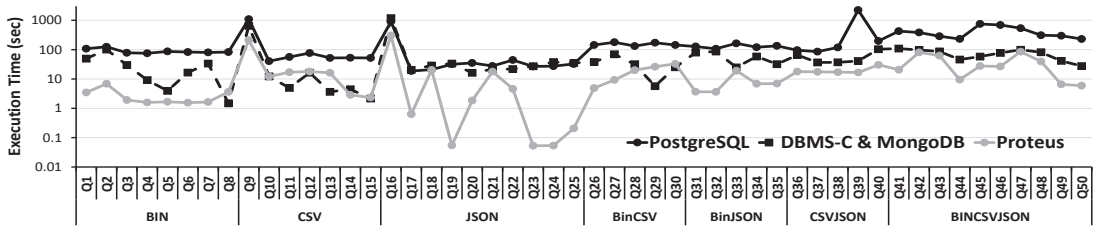


Figure 4.14 – For a spam analysis workload, Proteus outperforms the other systems in the majority of queries due to i) its lightweight, specialized-on-demand code paths, and ii) the caches it builds as a side-effect of query execution.

country), and the bot responsible for it. These files are the input to the data mining workflows of Symantec; classification and clustering are performed over them, through which each mail is assigned to a class per classification criterion. In every iteration of the workflow, output is stored in comma-separated-values (CSV) files containing an identifier of each e-mail, various assigned classes, etc. Finally, data is transformed and loaded in an RDBMS, with the use of which further calculations are made. This process is repeated for every new batch of JSON files: In each repetition, "fresh" JSON and CSV files have to be loaded in a DBMS and queried along with pre-existing data.

Analyzing this data involves queries over combinations of the datasets. We compare three possible solutions, for which we use i) an RDBMS that has been extended to support richer data models, ii) an RDBMS for flat data and a document store for hierarchies, and iii) Proteus, which reshapes itself based on each query. For approach I, we use PostgreSQL because it utilizes the most efficient JSON encoding out of the general-purpose systems we tested. For approach II, we use the combination of the specialized systems DBMS C and MongoDB, along with a mediating layer on top of them to facilitate cross-format queries and data exchange.

The input comprises a 20GB JSON file of 28M objects with arbitrary field order, a 22GB CSV file of 400M records, and a 95GB database table of 500M records. PostgreSQL and MongoDB load the JSON data prior to querying it. PostgreSQL requires 22GB to store the binary JSON encoding, and MongoDB requires 30GB. Proteus builds a structural index during its first access to the JSON file; its size is ~ 24% of the file. DBMS C and PostgreSQL load the CSV data prior to querying it. Proteus again builds a structural index during the first access, storing the position of every 5th field; its size is ~ 17% of the file. Regarding binary storage, Proteus operates over binary column files similar to the ones of MonetDB. Proteus caching is enabled in this experiment. The experiment starts with the OS cache containing the binary table, and none of the systems having accessed the CSV nor the JSON data yet.

We launch a workload of 50 queries sequentially, and progressively query a variety of the datasets. The queries perform selections, 2- and 3-way joins, unnests of JSON fields, result groupings, and aggregate computations; Appendix A.1 has more details on the queries. Projectivity ranges from 1 to 9 fields, and selectivity from ~ 1% to 25%. We group together queries accessing the same datasets. We show the results in Figure 4.14.

Q1-Q8 touch the binary dataset. For Q1-Q7, Proteus is the fastest approach, which corroborates the findings for selection and grouping queries over TPC-H data. Q8 has a very selective predicate on the field used by DBMS C to sort the input data, therefore DBMS C skips a large part of the dataset and is slightly faster than Proteus.

Q9-Q15 touch the CSV dataset. For DBMS C and PostgreSQL, the execution time of Q9 includes the loading time of the CSV dataset. Proteus answers queries over the original data, also building a structural index during Q9 and caching any fields it converts to answer the query. Q9 takes Proteus 440 seconds less than DBMS C, and 880 seconds less than PostgreSQL. DBMS C is faster than Proteus in Q11 because it operates over binary data, whereas Proteus converts data fields on-the-fly and pays to cache them for further use. Indeed, Proteus partially serves Q12 from its caches. On the other hand, Q12 also has a filtering predicate on a string field. Proteus opts not to cache string fields, whereas DBMS C performs dictionary encoding of string values during loading and exploits it in Q12; still, both systems have similar performance. Q13 is also heavy on string-based operations, which explains why DBMS C is faster. For Q14 and Q15, Proteus is the fastest approach because of the binary caches it populates and its minimal generated engine.

Q16-Q25 touch the JSON dataset, so MongoDB becomes active. For Q16, all systems behave as in Q9: Proteus exploits that the JSON dataset is accessed for the first time and caches data aggressively since the caching cost is masked by I/O and the structural index construction. Q16 takes Proteus 600 seconds less than MongoDB, and 800 seconds less than PostgreSQL. For Q17, Proteus uses its caches to speed-up execution significantly. For Q18 and Q21, caches are less useful because the queries involve string fields, which Proteus extracts and processes from the JSON file at query time. Using a policy of caching strings would benefit Proteus in the short term, but it would also pollute the caches with string objects. Still, Proteus is slightly faster than the other systems. For the rest of the queries, custom code generation combined with judicious data conversions and adaptive caching make Proteus faster.

Q26-Q30 join binary and CSV data. The materialization overhead of DBMS C is insignificant because these queries are very selective. Still, Proteus is faster for Q26 and Q27. Likewise, it is barely noticeable for Proteus that Q28 includes predicates on string fields of the CSV file. DBMS C is faster for Q29 because Proteus again has to access a string field in the CSV file, and at the same time DBMS C skips multiple data entries because of a filtering predicate on its sorting key. In general, both DBMS C and Proteus offer competitive performance for this query range. Finally, Q31-Q35 join binary and JSON data, Q36-Q40 join CSV and JSON, and Q41-Q50 join all three datasets. Q39 is very expensive for PostgreSQL because it picks a sub-optimal, nested-loop-based plan. Proteus is consistently the fastest system for two reasons: First, as discussed in Section 4.7.1, customizing the query engine gives significant performance benefits. Second, Proteus adaptively caches accessed values, thus after some point it largely operates over its binary caches, instead of the verbose CSV and JSON datasets.

	Load CSV	Load JSON	Middle-ware	Q39	Queries (Rest)	Total
PostgreSQL	1019	792	0	2226	7468	11505
DBMS-C & MongoDB	711	1067	43	29	1810	3660
Proteus	0	0	0	17	1231	1248

Table 4.2 – Execution time per Symantec workload phase.

At the end of the workload, the cache size for the CSV data is ~ 30% of the CSV file. The cache size for the JSON data is only ~ 2.5% of the JSON file. JSON caches are more compact because although the number of CSV and JSON fields of interest is almost the same, the JSON file contains 28 million verbose JSON objects to be partially cached, whereas the CSV file contains 400 million narrow tuples. Interestingly, the JSON caches are more impactful for the workload because of the increased access cost for the JSON dataset. Therefore, if we were to drop any caches to adhere to a tighter memory budget, we would start from the ones for CSV data.

Aggregate Performance. Table 4.2 presents the accumulated execution time spent in each workload step. Proteus is 9.12× faster than using an RDBMS with added support for richer data models (PostgreSQL) and 2.9× faster than the approach of packaging together multiple query engines and using the appropriate one for each specialized scenario (DBMS C & MongoDB). We isolate Q39 because it is an outlier for PostgreSQL that highlights the problem of extending existing systems without deeply integrating support for the added data models and formats. Q39 performs a join between the CSV and JSON datasets. PostgreSQL, however, treats JSON data as a BLOB-like datatype, which is essentially opaque to its optimizer. The result is that the optimizer chooses an expensive nested-loop join. If we exclude Q39 from the aggregated execution time, Proteus is still 7.4× faster than PostgreSQL. Finally, even if we focus completely on execution time and exclude any other overheads from the workflow (e.g, data loading cost, overhead of middleware layer), Proteus still is the fastest system overall.

In conclusion, Proteus flexibly accesses a real-life workload of heterogeneous datasets while being as fast as a specialized system per use case. Besides being fast regardless of its input, Proteus achieves an additional speed-up by adapting to the workload through caching structures built as a side-effect of querying.

4.8 Summary

Data analysis solutions over heterogeneous data have always involved a trade-off: be flexible and serve diverse datasets at the cost of performance, or be rigid and specialized for a specific scenario, thus leading users to employ a different system per use case.

This work presents a system design that offers flexibility to users, exposing heterogeneous datasets under a single interface, while also exhibiting the response times of a system specialized per use case. The design couples i) an expressive query algebra that masks data heterogeneity with ii) on-demand customization mechanisms that produce a new system

implementation per query. Based on this design, we build Proteus, a query engine that natively supports CSV, JSON, and relational binary data, and also specializes its entire architecture to each query and the data that it touches via code generation. Proteus also customizes its caching component, specifying at query time how these caches should be shaped to better fit the overall workload.

Proteus serves synthetic and real-world workloads efficiently: it outperforms state-of-the-art open-source and commercial approaches without being tied to a single data model or format, all while operating transparently across heterogeneous data. The ability of Proteus to dynamically specialize itself opens multiple opportunities for further optimizations.

5 Big Data Virtualization

The typical enterprise data architecture consists of several actively updated data sources (e.g., NoSQL systems and data warehouses), and a central data lake such as HDFS, in which all the data is periodically loaded through ETL processes. To simplify query processing, state-of-the-art data analysis approaches solely operate on top of the local, historical data in the data lake, and ignore the fresh tail end of data that resides in the original remote sources. However, as many business operations depend on real-time analytics, this approach is no longer viable. The alternative is hand-crafting the analysis task to explicitly consider the characteristics of the various data sources and identify optimization opportunities, rendering the overall analysis non-declarative and convoluted.

We design *System-PV*, a real-time analytics system that masks the complexity of dealing with multiple data sources while offering minimal response times. System-PV extends Spark with a sophisticated data virtualization module that supports multiple applications – from SQL queries to machine learning. The module features a query compiler that considers source complexity and location (i.e., "local" and "remote" sources), and a two-phase optimizer that produces and refines the query plans, not only for SQL queries but for all other types of analysis as well.

5.1 Introduction

In the past decade, there has been an explosion in terms of data volume and variety, as well as in terms of demand for data-driven insights. Daily business operations are supported by a diverse set of applications, each with its own characteristics. Therefore, different parts of the same organization end up using different systems depending on their application requirements. NoSQL stores and OLTP systems are widely used as operational stores that store the most recent data as generated by customer transactions, user tweets, etc. ETL processes are periodically run over each operational data source to extract the data, transform it appropriately, and load it in a unifying data lake, such as HDFS, or a relational data warehouse, on top of which various types of analytics are performed. Querying data in such complex ecosystems is a significant challenge.

SQL-over-Hadoop: Ignore problem, or hand-code solution. Data analysts typically use a scale-out processing system, such as Spark [260], to run analytics over the data portion stored in the data lake. A major problem of accessing only the data lake is staleness, as the *tail end* of data (i.e., most recent and interesting data [23]) in the operational sources is ignored. Data staleness is often unacceptable because many applications require analysis of the tail end of the data, as well as the historical data.

To facilitate analysis over multiple data sources, engines such as Spark [260] and Hive [238] offer connectors [43, 198] to provide access to data sources that are external to the data lake. Although the connectors provide the basic mechanism to access external sources, the data analysts carry the burden of efficiently using them.

SQL-over-Hadoop: Example. A user who creates a Spark job to process both the historical data in the data lake and the most recent data in the external sources has to hand-code her analysis using low-level logic that considers the following factors: 1) the location of data as well as recent data updates in the external sources, 2) potential ETL invocations to ingest data into the lake, 3) the data overlap between the external sources and the data lake, 4) potential schema mediation between data sources, 5) optimization opportunities for the overall analysis. Going through multiple steps and writing boilerplate code before launching any type of analysis is a non-sustainable, complicated process.

Data federation systems are an established alternative for queries over multiple sources, yet they have two shortcomings that hinder their use in modern applications: First, traditional federation systems focus solely on SQL analytics. Second, they encounter difficulties optimizing queries over logical datasets that are physically spread across the data lake and an external source, and therefore exhibit suboptimal performance [35]. Thus, users end up compromising data freshness by operating only over the historical data in the data lake and ignoring the tail end of data in external sources.

Traditional Data Federation: Complexity leads to compromise. Data federation systems are an established alternative for queries over multiple sources, yet they have two shortcomings which hinder their use in modern applications: First, traditional federation systems focus solely on SQL analytics. Second, they encounter difficulties optimizing queries over logical datasets that are physically spread across the data lake and an external source, and therefore exhibit suboptimal performance [35]. Thus, users end up compromising data freshness by operating only over the historical data in the data lake and ignoring the tail end of data in external sources.

Polymorphic Virtualization. This work designs a *data virtualization module* that provides a unified view over multiple data stores that are heterogeneous in terms of i) data model, ii) update rates, and iii) query capabilities. The design enables ***p**olymorphic **v**irtualization*, i.e., masking the complexity of dealing with multiple stores, while offering minimal response times [144].

To abstract away the complexity stemming from data source variety, the data virtualization module exposes a global schema on top of logically contiguous datasets that are physically partitioned across systems. The module then uses a *location-aware compiler* to map the analysis from the global virtual schema to the actual sources.

The module additionally uses a *two-phase optimizer* to optimize the overall analysis and offer minimal response times. The optimizer operates in two phases to optimize both SQL and general analysis tasks, and to reduce the overall complexity of query optimization over multiple sources. Phase I considers established cost-based query optimization techniques for complex SQL queries, without being cluttered by the details of dispersed data sets. Phase II optimizes all types of data analysis by considering the properties of the underlying data sources to generate an efficient execution plan.

We validate our design by coupling the data virtualization module with the Spark framework to implement *System-**PV***. System-PV maintains all the Spark APIs and thus can support all types of Spark applications (e.g., OLAP, machine learning, etc) over a virtual, simplified schema. The location-aware compiler of System-PV rewrites a data analysis program into a Spark script over the actual physical schema. The two-phase optimizer rewrites the resulting script using the sophisticated IBM Big SQL™ [119] query optimizer for its SQL-oriented Phase I, and the Spark SQL Catalyst optimizer [43] for its universal Phase II. As a result, System-PV efficiently serves a spectrum of choices for enterprise applications, from operating on stale data that is in the data lake, to accessing data remotely in place, as well as a combination of the two by allowing data sets to be split between the data lake (i.e., the historical part) and a remote data source (i.e., the tail end of fresh data), all while masking the actual data source and schema complexity from the users.

Overall, the work in this chapter makes the following contributions:

- We identify shortcomings of the state-of-the-art systems when deployed on top of data lake environments and accessing fresh data in external data sources (Section 5.3).
- Motivated by the challenges that users face, we design System-PV, a real-time analytics system that extends Spark by introducing a data virtualization module that employs a location-aware compiler and a powerful two-phase optimizer. System-PV supports and optimizes diverse analytics over a global virtual schema that masks data source variety and complexity (Sections 3-5).
- We evaluate System-PV using the TPCx-BB [20, 114] dataset appropriately extended to incorporate non-relational data, and show that System-PV is faster than Spark when accessing multiple data sources, often by more than an order of magnitude. Further, System-PV considers fresh data in external data sources at negligible performance overhead compared to operating solely on top of the data lake, while abstracting away the complexity from the user (Section 6.7).
- We provide insights based on our experiences operating in data lake settings (Section 5.8).

5.2 Related Work

System-PV leverages decades of research in database views, ETL, and data federation systems [71, 75, 243, 244]. This section surveys these works and highlights how System-PV pushes the state-of-the-art further.

Querying Multiple Sources. In recent years, scale-out frameworks, such as Spark [43], Pig [198], and Hive [238], offer specialized connectors to allow queries over multiple data sources that are "external" to HDFS (e.g., RDBMS), yet lack higher-level abstractions to hide source complexity. In addition, even when such systems perform cost-based optimizations [2], their optimizers ignore external source characteristics.

On the contrary, traditional data federation approaches have extensively studied query execution across multiple data sources [217, 216, 124, 239, 75]. However, these approaches focus solely on SQL-based data analysis and lack support for iterative or other kinds of analytics (e.g., machine learning). In addition, federated optimizers encounter difficulties when producing plans for queries that touch datasets split across multiple sources; deciding the optimal way to execute a query with multiple JOIN and UNION ALL operations over different data sources is non-trivial [35]. Therefore, users have been avoiding such scenarios.

System-PV introduces a two-phase optimizer to specifically target cases with complex relationships between data sources, thus allowing a single logical dataset to be split across different sources, and handling data overlap. As we show later, such data distributions are frequent in data lake settings due to the periodic nature of ETL processes. Two-phase optimization was initially proposed as a way to perform site selection at runtime, and thus balance the load equally among the execution sites [72]. Then, the XPRS parallel DBMS [130] employed two-phase optimization to reduce the overall search space of possible parallel query plans. Garofalakis et al. proceeded to provide a formal framework for reasoning in terms of both single- and two-phase optimization [112]; the framework uses metrics such as the "critical path length" of a parallel query plan, the amount of resources that an operator reserves, and the estimated execution time of an operator. Two-phase optimization can result in a final physical query plan that is different from the optimal plan [156]; still, combining a two-phase optimizer with sufficient information about the overall physical database design generally results in efficient distributed query plans [94].

Polystores. Another method to serve diverse types of queries over heterogeneous data sources is through *polystore* systems [28, 65, 95, 99, 164] that bundle together multiple query engines and use the most appropriate per query type. Polystore systems apply frequent and multi-directional data migration across the various engines [99]. Data exchange among multiple systems is challenging because it i) complicates query optimization and ii) requires connecting each system with every other system via specialized pairwise connectors [177]. The Myria [251] system uses the architecture of a federated database system as its blueprint and operates over a polystore environment. Myria uses an extended relational, rule-based optimizer, whose rules allow expressing complex operations in ways supported by different backends. In addition,

Myria uses PipeGen [126] – an underlying communication framework – to facilitate data transfer between the different backends it supports. PipeGen reduces data transfer cost by allowing data stores to exchange Apache Arrow [21] binary buffers.

Still, data transfers to and from operational data stores create additional load that can affect the stability and performance of the data stores: As opposed to polystores, we design System-PV for scenarios where the majority of data is stored in the data lake and only the tail end of the data is in external sources. In such environments, data is typically transferred from the external sources to the data lake; unidirectional communication avoids overloading the operational stores and reduces the number of plans that the optimizer considers.

ETL. ETL (Extraction, Transformation, Loading) [170, 243] is a process that populates a data warehouse with data originating in external sources. In recent years, HDFS is frequently used as the staging/destination area [211]. The popularity of HDFS has led to specialized tools [7, 10] for data ingestion. System-PV performs ETL on demand when accessing a variety of external sources and masks ETL costs through data-source-specific optimizations.

Database Views. Database views are frequently used to mask the underlying structure of the data. System-PV supports both lazily evaluated and materialized views depending on the user requirements and the optimizer guidelines. Views are also extensively used in the domain of data integration [165], where data sources are mapped to a global schema using *local-as-view* (LAV [152]) or *global-as-view* (GAV [75]) methods. System-PV uses the GAV variation to form a global virtual schema.

5.3 Motivation and Background

We now use an example to describe the challenges faced by users when developing applications that access external sources. We use Spark as a representative state-of-the-art framework [260]. Spark is frequently deployed in data lake environments because it supports various types of data analysis (e.g., OLAP, machine learning, etc.) and is compatible with various types of external sources. Spark provides both a procedural (e.g., Scala) and a declarative interface through Spark SQL [43]. Other frameworks (e.g., Hadoop [8], Hive [238], and Flink [38, 6]) have similar characteristics, and their users face similar challenges.

Motivating Example. Figure 5.1 depicts a modern data analysis scenario: A company uses an RDBMS to store transactional data about product sales (*Sales* dataset), and a NoSQL key-value store to store the shopping cart data of online clients (*Shopping Carts* dataset). ETL processes periodically load the data into a central data lake (HDFS), over which users run analysis using Spark. Thus, the *Shopping Carts* dataset ends up being stored across the data lake (*CartsHDFS* table) and the key-value store (*CartsKV* table). Similarly, the *Sales* dataset is spread across the data lake (*SalesHDFS* table) and the RDBMS (*SalesFact*, *Products* tables). The *Products* table is a dimension table, which is frequently updated and thus remains in the RDBMS through its entire lifetime. On the contrary, the shopping cart data and the fact table of the sales data

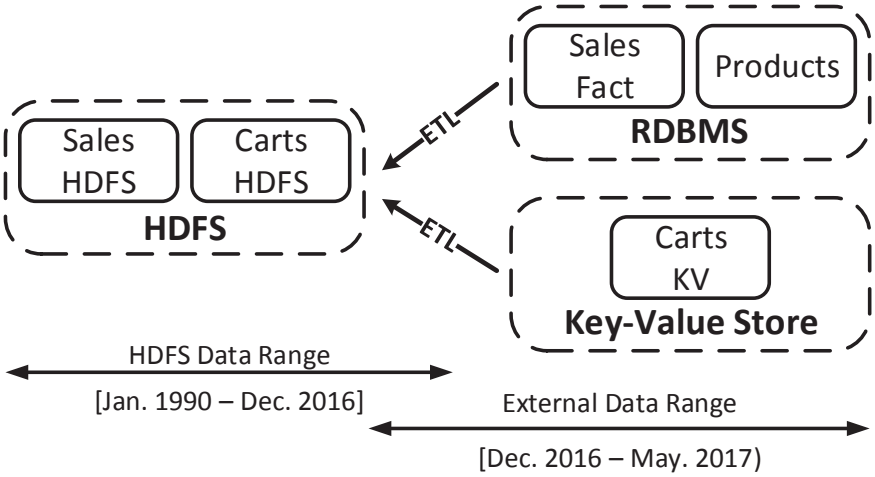


Figure 5.1 – Typical scenario in a data lake: Analyzing recent, actively updated data along with historical data.

are periodically loaded to the data lake, while new data is continuously appended into their RDBMS and key-value store parts, respectively. Thus, the *tail end of the data* resides in the external sources.

Listing 5.1 shows a Spark SQL query that computes the number of products that were placed in customer shopping carts and eventually purchased. The query performs a join between the *Sales* and *Shopping Carts* datasets, followed by an aggregation (Lines 16-18). Putting together the query script is non-trivial because of numerous reasons. First, a single logical dataset (*Sales* and *Shopping Carts*) consists of subsets that are physically stored across the data lake and an external data source. Thus, the user needs to be aware of the portions of these datasets that are present in each source, then manually perform the necessary filter operations to extract the correct data from each source (Lines 2, 6, 10, 12), and finally perform the appropriate union operations (Lines 8, 14). Second, these subsets might overlap. In our example, the sales data of December 2016 is stored in the data lake but is still actively updated in the RDBMS (e.g., for auditing reasons); likewise for the carts data. Thus, the user must consider her desirable query semantics in order to determine where to read the data from. In this example, the user wants to get the most recent data values and thus must be careful to read the data corresponding to December from the external sources instead of HDFS (Lines 2, 6, 10, 12). Third, the physical data layouts of subsets of the same dataset can differ. For example, the part of the *Sales* dataset in the RDBMS is normalized across two tables (*SalesFact*, *Products*), whereas the subset stored in the data lake is denormalized (*SalesHDFS*). Thus, the user must join the *SalesFact* and *Product* tables (Line 6). Note that this is not the case when retrieving the sales data from the data lake (Line 2). Finally, when loading the data in the lake, the ETL process might perform lightweight data transformations, which must be taken into account when querying the data (not shown in this example).

```
/* HDFS side of Sales dataset */
SalesHDFS.filter("sold_date < 20161201")
/* RDBMS, normalized side of Sales */
SalesDB = SalesFact.join(Products,
      SalesFact("s_id")===Products("s_id"))
SalesDB = SalesDBAll.filter("sold_date >= 20161201")
/* Unified Sales dataset */
Sales = SalesHDFS.unionAll(SalesDB)
/* HDFS side of Carts dataset */
CartsHDFS.filter("sold_date < 20161201")
/* NoSQL side of Carts dataset */
CartsKV.filter("sold_date >= 20161201")
/* Unified Carts dataset */
Carts = CartsHDFS.unionAll(CartsKV)
/* Get number of products placed in shopping carts and eventually purchased */
query = Sales.join(Carts,
                Sales("user_id")===
                Carts("user_id")).count()
```

Listing 5.1 – Spark SQL query across multiple sources.

```
query = VirtualSales.join(VirtualCarts, VirtualSales("user_id") ===
    VirtualCarts("user_id")).count()
```

Listing 5.2 – System-PV query across multiple sources.

As queries become more complex, the burden on the user increases; she has to hand-code more complex analysis plans, all while considering the desirable query semantics, potential data overlap, diversity in terms of data layouts, etc. In addition, every time the user wants to submit a new query, she must consider whether any of her previous assumptions have changed. Thus, query formulation over intermingled data sources becomes complex and non-declarative. On the contrary, System-PV masks source complexity by exposing a virtual schema; Listing 5.2 shows the System-PV query corresponding to the Spark SQL query of Listing 5.1. The System-PV query is significantly simpler than the Spark SQL equivalent; we will be discussing this query in detail later.

The Spark Computing Framework. We now provide a brief overview of the Spark computing framework since System-PV builds on top of it. Spark supports various types of applications (e.g., OLAP and machine learning) written as Scala, Java, and Python scripts, or as declarative queries through Spark SQL [43]. The architecture of Spark SQL is depicted in Figure 5.2a. Spark SQL manipulates *DataFrames*, which are distributed collections of structured records. Users express their analysis through a combination of procedural code that invokes the *DataFrame API* and declarative SQL queries that are translated to *DataFrame API* calls by Spark SQL. Regarding data access, the *Data Sources API* enables access to common HDFS formats (e.g, Avro [1], Parquet [9], etc.) and to external sources, such as RDBMSs and key-value stores. Adding support for an additional data source only requires coding in a plug-in that implements the *Data Sources API*.

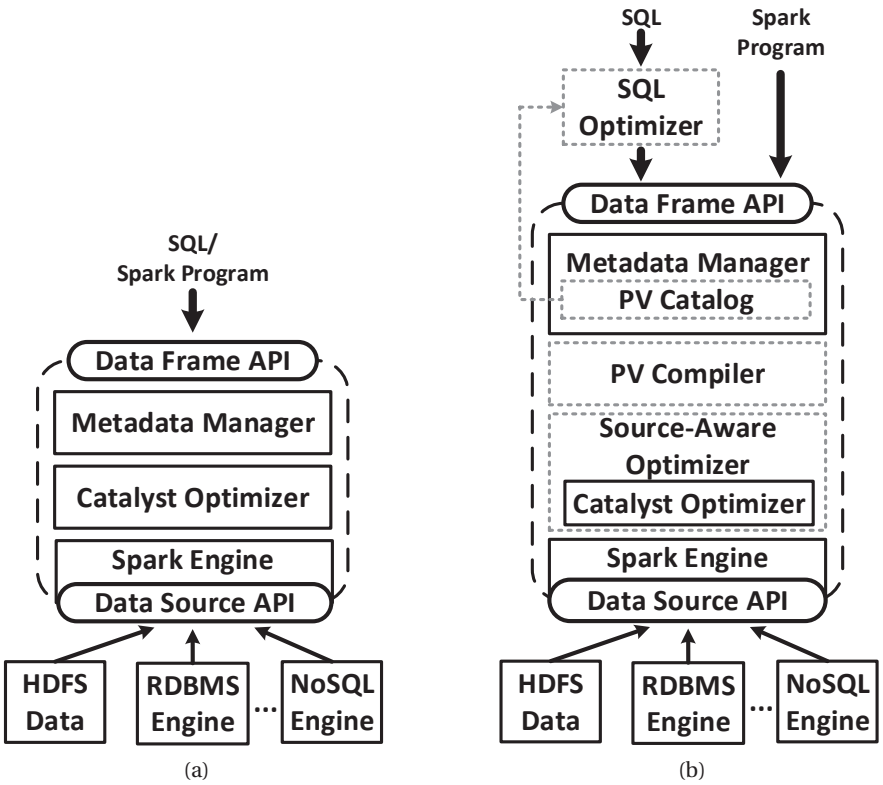


Figure 5.2 – Architecture of (a) Spark SQL and of (b) System-PV. Dotted boxes in (b) represent extensions.

External tables and user-created *DataFrames* can be registered in the Metadata Manager (e.g, Hive Metastore [238]). Once an SQL query arrives, Spark rewrites it to the *DataFrame API* and optimizes it using the *Catalyst* optimizer. *Catalyst* currently performs logical rewrites (e.g., filter pushdown) and basic cost rewrites (e.g., choosing between a broadcast and a shuffle join). Spark pushes computation to external sources when applicable. Finally, the query engine of Spark executes the resulting physical plan.

5.4 System-PV

System-PV addresses the challenges related to data analysis over multiple data sources by making the following two key contributions: First, System-PV abstracts away the complexity of writing data analysis applications through a *data virtualization module* that exposes a "virtual" schema across heterogeneous data sources while still supporting all types of Spark applications. Instead of forcing the user to manually deal with data locations, ETL processes, data overlap, and conflicting schemata across sources, System-PV operates on top of *view definitions* that mask the complexity of the underlying data sources. Second, System-PV optimizes the execution of data analysis scripts using a powerful *two-phase optimizer* that supports both

SQL and arbitrary analysis scripts, performs cost-based optimizations, and also considers the properties of the external sources. As a result, System-PV offers the performance of hand-coded, fine-tuned execution plans, while providing a declarative way to perform data analysis across multiple sources.

We build System PV on top of Spark, because Spark i) supports a wide range of analysis types and ii) is extensible in terms of supported data sources. System-PV serves both SQL queries as well as arbitrary data analysis scripts (typically machine learning jobs) expressed using the Spark *DataFrame API* over the global virtual schema. Figure 5.2b presents the high-level architecture of System-PV. The *SQL Optimizer* optimizes SQL queries. Arbitrary data analysis scripts are passed directly to the *PV Compiler*. System-PV uses the IBM Big SQL™ [119] query optimizer to optimize the incoming SQL queries, which is based on the IBM DB2™ query optimizer, and is more sophisticated than Catalyst as it considers cost-based optimizations as well as additional query rewrite opportunities. The output of the optimizer is an optimized SQL query plan over the virtual schema, which is then expressed in the Spark *DataFrame API* and routed to the *PV Compiler*.

The *PV Compiler* rewrites the query plan in a form that references the original data sources and is understood by the Spark Engine. The PV Compiler uses the view definitions that comprise the virtual schema and are contained in the *PV Catalog*. In particular, the PV Compiler replaces each view occurrence with a sub-plan corresponding to its definition, producing an extended plan over the external data sources.

After the compilation phase, the *Source-aware Optimizer* performs a series of logical rewrites to the plan. We implement the Source-aware Optimizer as an extension of the Spark Catalyst Optimizer. Its responsibility is examining the underlying data sources and producing plans conforming to their capabilities. For example, the Source-aware Optimizer detects whether the data source targeted is an RDBMS or a NoSQL key-value store, and rewrites the logical plan accordingly. The output is a physical plan that the Spark Engine executes.

The following two sections elaborate on the System PV components: Section 5.5 explains how to express a virtual schema over the different data sources and launch analysis over the schema. Then, Section 5.6 presents the two-phase optimization process that System PV follows in order to optimize the overall analysis.

5.5 Compiling Cross-store Queries

System-PV users develop analysis scripts over a global virtual schema that abstracts away the complexity of the underlying data sources. We now discuss the properties of the virtual schema and describe how System-PV automatically rewrites user programs over the virtual schema into specialized programs that reference the external sources.

```
Scan(srcName)
Select(expression,view)
Project(expression,view)
Join(expression,view1,view2)
Union(view1,view2)
UDFunc(expression,view)
Materializer(expression,view)
```

Table 5.1 – Operators used in the view definitions of System-PV.

5.5.1 Exposing a Virtual Schema

The virtual schema consists of view definitions over datasets that are scattered across various data sources. A view provides an abstraction over a logical dataset that is physically stored in one or more data sources. We now discuss the characteristics of the view definitions.

Data Sources. System-PV supports views over both "native" and external sources. Specifically, it supports "native" Spark storage (i.e., Parquet files [9], transient in-memory *DataFrames*, and *DataFrames* cached in Tachyon [167]) and external sources such as RDBMSs and key-value stores. System-PV connects to an external source by invoking the Spark *Data Sources API*.

View Definitions. In most System-PV use cases, the view definitions that comprise the global virtual schema are *created once*; users then submit queries over the virtual schema. Note that the views need not be materialized.

To express the views, System-PV uses a subset of the relational algebra and a number of user-defined scalar functions (UDFs) that correspond to lightweight ETL primitives. The algebra, which is presented in Table 5.1, is straightforward and allows composability of view definitions: a view can be defined based on a previously defined view. The algebraic operations take as input *views* and *expressions*. The *expressions* have different semantics depending on the operation. In the case of `Select` and `Join`, the expression filters the result, whereas in the case of `Project`, the expression projects certain columns of the dataset. `UDFunc` is an aggregating term for the various UDFs that correspond to lightweight ETL processes. Finally, a `Materializer` produces a materialized view. Depending on the value of the `mode` parameter, the view is cached as a Parquet file, a *DataFrame* stored in memory, or a *DataFrame* stored in Tachyon [167].

Listing 5.3 shows the view definitions for our running example, which are created once. Using the view definitions, the users operate directly on the virtual schema (*VirtualCarts, VirtualSales)* and thus can be unaware of the actual data locations. Listing 5.2 shows the simplified System-PV query over the virtual schema that corresponds to the Spark SQL query of Listing 5.1.

```
1 cKVSel = Select('t >= 20161201', Scan(CartsKV))
2 cHDFSSel = Select('t < 20161201', Scan(CartsHDFS))
3 VirtualCarts = Union(cKVSel,cHDFSSel)
4 SalesDB = SalesFact.join(Products,
5     SalesFact("s_id")===Products("s_id"))
6 sDBSel = Select('t >= 20161201', Scan(SalesDB))
7 sHDFSSel = Select('t < 20161201', Scan(SalesHDFS))
8 VirtualSales = Union(sDBSel,sHDFSSel)
```

Listing 5.3 – Views for running example, created once.

Managing Views. System-PV contains a catalog service, namely *PV Catalog*, to maintain the virtual schema. Apart from storing the view definitions, the PV Catalog captures information about each data source, such as its type and capabilities (e.g., whether the data source exposes an index or whether it supports range queries).

Whenever an ETL process loads new data in the data lake, System-PV updates automatically the view definitions in the PV Catalog. For this purpose, System-PV assigns a "watermark" to the views that capture a certain temporal range (shown in blue in Listing 5.3). Additionally, System-PV assigns a temporal range to each data batch loaded from the external sources to the data lake; these data batches are stored as separate HDFS partitions [155]. The range of a data batch corresponds to the period from the transaction time[1] of the oldest batch entry to that of the newest batch entry. In the example of Figure 5.1, loading the tail end of data into the data lake would result in a batch with the range `[Dec.2016 - May 2017)`. System-PV supports external sources that handle transactional workloads, such as RDBMSs or key-value stores. If the last batch ingested into the data lake corresponds to the range $[t_1, t_2)$, then System-PV automatically assigns the range $[t_2, +\infty)$ to the data in the external source. When an ETL process loads a data batch, it edits the watermarks of the affected views to incorporate the temporal range of the incoming batch, thus triggering System-PV to update the view definitions.

Data Overlap. A common scenario is to have a large portion of a dataset stored in the data lake whereas the tail end of the data is stored in an actively updated external source. Depending on the nature of the application and the periodic ETL processes, it is possible that these two subsets overlap. In the example of Section 5.3, the sales data corresponding to the period between 1990 and 2016 is archived in the data lake (HDFS). The data for December 2016, however, is also stored in the company's operational data store because updates still occur over this data. This data will eventually be pushed to the data lake and the stale HDFS counterpart will be refreshed. Until then, System-PV enables users to define a view that specifies which side (HDFS or the RDBMS) should serve the overlapping data. This view is defined based on the application requirements: If data freshness is important, then the data corresponding to December 2016 must be fetched from the RDBMS as shown in Line 6 of Listing 5.3. Otherwise, accessing the local HDFS data is prone to be more efficient.

[1] The time when the fact is (logically) current in the database [229].

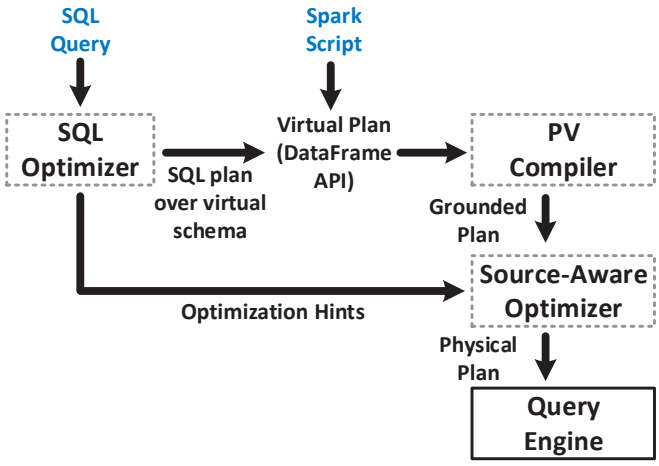


Figure 5.3 – System-PV Pipeline.

Lightweight ETL. System-PV handles datasets that are physically split across the data lake and an external source even when the corresponding data subsets have different schemata. Specifically, System-PV offers primitives for expressing lightweight ETL processes. Users can remap schemata by changing the name, datatype, and the order of fields. In addition, users can employ UDFs that transform column values in order to, for example, convert different units of measurement and handle out-of-bound values (e.g., negative ages). System-PV also handles more complex cases, such as the one presented in Figure 5.1, where the *Sales* data is normalized in the RDBMS but it is denormalized in the data lake. As shown in Line 4 of Listing 5.3, users can express views using join operations to denormalize the external data at query time.

5.5.2 Querying over a Virtual Schema

As shown in the example of Listing 5.2, System-PV users express their scripts directly over a virtual schema. At some point, System-PV must therefore translate the virtual schema to the actual heterogeneous data sources. Figure 5.3 shows how PV Compiler performs the translation: When a user expresses an SQL query or a procedural script, System-PV generates a logical plan over the virtual schema that is described using the *DataFrame API*; we call this a *virtual plan*. Then, System-PV feeds the *virtual plan* to the *PV Compiler*, which in turn uses the view definitions stored in the PV Catalog to rewrite the plan into a *grounded plan*; the grounded plan references the original data sources and is understandable by the Spark engine. The *virtual* and *grounded* plan corresponding to our running example are depicted in Figure 5.4.

Specifically, the PV Compiler traverses the *virtual plan* until it locates scan operations corresponding to virtual datasets. For each of the virtual datasets, the PV Compiler looks up its view definition in the PV Catalog, and outputs code that describes how to access the corresponding data in the external data sources. The PV Compiler performs the rewriting using two

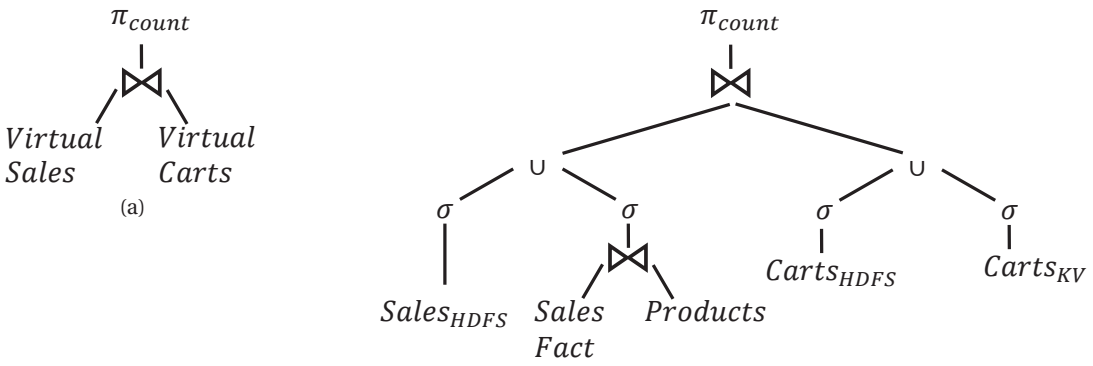


Figure 5.4 – Virtual plan of our running example (a), and its corresponding grounded plan (b).

components: an *Algebraic Rewriter* and an *Expression Rewriter*. The Algebraic Rewriter takes as input a view definition, maps the operators of the view into equivalent operations of the *DataFrame API*, and calls the Expression Rewriter to transform expressions when necessary. For a view defined as `Select('x < 10',Scan(table))`, the Algebraic Rewriter invokes the Spark SQL `filter()` function, and the Expression Rewriter produces the code for the predicate evaluation.

Most of the algebraic nodes of Table 5.1 have 1-1 mappings to Spark operations, similar to the ones of `Select`. System-PV models `UDFunc` operations as overloaded versions of a projection operation. Finally, the `Materializer` operator is mapped to a different type of Spark operation (e.g., a persistent flush command, transient in-memory caching, etc.) based on a `mode` parameter specified at view definition time.

Summary. System-PV masks data source complexity by exposing a global virtual schema and by using a location-aware compiler to generate specialized scripts that access the external sources. System-PV also alters its view definitions to cater for ETL-triggered data updates.

5.6 A Two-phase Optimizer for Cross-store Analytics

System-PV allows users to perform their analysis over a global virtual schema, thus masking source complexity. Still, by enabling data analysis over a wide combination of heterogeneous data sources, the query optimization task seemingly becomes harder. Distributed query optimization is a well-studied problem [156], which is intensified in our case because i) System-PV supports different types of analysis besides relational queries and ii) heterogeneous data sources (e.g, NoSQL stores, RDBMSs) are accessed in the same analysis task.

System-PV makes use of a sophisticated two-phase optimizer. As shown in Figure 5.3, System-PV applies the first optimization phase to SQL queries only. System-PV applies the second phase regardless of whether the data analysis is expressed using SQL or an arbitrary Spark program. Specifically, when receiving an SQL query, System-PV applies the cost-based optimizations of a mature SQL optimizer by considering **only** the virtual schema (***Phase I***).

System-PV further optimizes the analysis plan by exploiting the capabilities of the underlying data sources (***Phase II***).

The IBM Big SQL optimizer performs numerous cost-based optimizations over an input SQL query, yet is unable to reason in terms of non-relational types of analysis such as Spark SQL procedural scripts. On the contrary, the Spark SQL Catalyst optimizer can process any type of analysis expressed in the Data Frame API – relational or not. Therefore, Phase I uses the specialized Big SQL optimizer so that it specifically target SQL analysis, and Phase II uses the Catalyst optimizer so that it is compatible and applicable to any type of Spark SQL analysis.

System-PV keeps the two optimization phases separate for two reasons: First, compared to optimizing procedural data scripts, optimizing declarative SQL queries is a more nuanced process, requires examining multiple execution plans, and typically benefits more from complex query optimization. Therefore, System-PV applies Phase I over SQL queries and not arbitrary data scripts. Second, the separation confines the universe of decisions in each phase. Unifying the two phases complicates plan enumeration: The source-specific rewrites of Phase II expand the query plan and thus increase the optimization space, so exposing the complexity to the SQL query optimizer would complicate its major task of identifying the appropriate join order.

5.6.1 Phase I: SQL Optimization

Optimizing SQL queries in a distributed setting is a challenging, error-prone task [31, 98, 124, 156, 173, 108]. In the case of Spark, the Catalyst optimizer is a promising first step, but at the time of writing, it mainly focuses on simple rewrites, and it supports very few cost-based optimizations. System-PV therefore uses the IBM Big SQL federated query optimizer because it supports sophisticated rewrites and cost-based optimizations.

As depicted in Figure 5.3, when System-PV receives an SQL query over the virtual schema, it routes the query to the SQL Optimizer. The SQL Optimizer requires data source information to perform costing and to come up with an efficient query plan; System-PV thus exposes such information for every “virtual table”, based on the metadata and statistics stored in the PV Catalog.

Specifically, when a dataset is split across an external source and the data lake, System-PV distinguishes between two cases. When the ETL process that loads the data in the data lake is frequent (i.e., when it exceeds a tunable threshold), the SQL optimizer considers only the data in the data lake. The tail end of data is thus masked during the Phase I optimizations and is considered only in Phase II. System-PV masks the tail end during Phase I for the following reasons: i) the tail end of data is typically small compared to the overall dataset, which is the default case in data lake environments, and ii) exposing more complex view definitions that capture the full dataset can complicate plan enumeration [35]. On the other hand, when ETL is sporadic, the SQL optimizer considers only the remote data source, since the remote data access dominates query execution costs.

System-PV exposes the local HDFS cluster to Big SQL as the "primary" data source, and the rest of the data sources as remote data stores. Depending on the data source exposed, the optimizer identifies the source capabilities (e.g., ability to perform projection pushdown, indexes) through specialized *source wrappers*[2] [71, 217]. Each source wrapper exposes data statistics to Big SQL to compute the overall query cost. Big SQL offers sophisticated, statistics-aware wrappers for RDBMS. On the other hand, Big SQL lacks a source wrapper for distributed key-value stores such as Cassandra [3]. System-PV therefore emulates the connection with an instance of Cassandra by re-using an existing wrapper: Specifically, given that Cassandra is a distributed key-value store, System-PV uses a wrapper designed for a parallel RDBMS, and informs Big SQL about a hypothetical hash index over the mock RDBMS to emulate Cassandra's key-based accesses. In addition, System-PV specifies a data partitioning scheme that the mock RDBMS hypothetically uses (e.g., hash partitioning) to emulate the partitioning scheme employed by Cassandra [4]. Finally, System PV collects statistics over Cassandra and injects them in the PV Catalog. Overall, System PV uses the different source wrappers of Big SQL and the accumulated data statistics to make well-informed decisions for SQL query optimization.

The SQL Optimizer uses the information of the exposed data sources to produce an optimized logical query plan over the *virtual schema*. In addition, it produces information about the corresponding physical plan. For example, the optimizer indicates the physical join algorithms to be used, and potential intermediate result materializations. System-PV uses the information about the physical plan as optimization hints during the source-aware optimization that produces the final physical plan (Phase II).

The optimized logical query plan is forwarded to the PV Compiler, which rewrites any occurrences of views and generates the *grounded plan* that references the original data sources. The *grounded plan* along with the optimization hints are then passed to the Source-aware optimizer used in Phase II.

5.6.2 Phase II: Source-aware Optimization

The second optimization phase applies source-specific optimizations to all data processing tasks, regardless of whether they are expressed in SQL or procedural code, through use of the *Source-aware Optimizer*.

An issue of Catalyst is that it misses multiple optimization opportunities for queries over external sources. Specifically, Catalyst uses the *Data Source API* to access external sources. The *Data Source API*, however, is meant for single-table accesses. As a result, only selections

[2] The data source wrappers of Big SQL are not to be confused with the data source connectors of the Spark Data Source API; the former are used during query optimization, whereas the latter only perform data access during query execution.

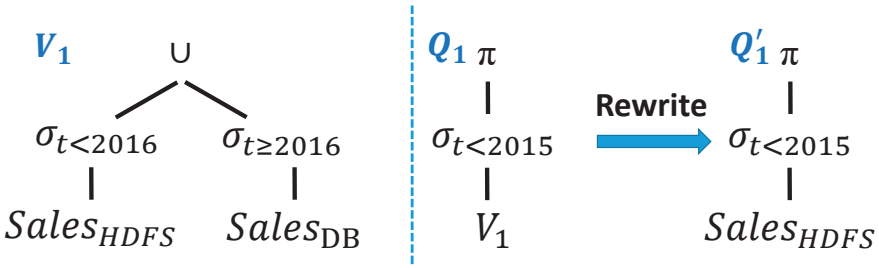


Figure 5.5 – Query plan simplification during source-aware optimization.

and projections are pushed down to the external sources. More complex operators such as joins are not pushed down, thus often missing opportunities for reducing the network traffic. Even worse, when the underlying external source is not an RDBMS, very few operations are pushed down.

As shown in Figure 5.3, the input of the *Source-aware Optimizer* comprises i) a *grounded plan* that references the original data sources, and ii) the optimization hints produced by the SQL optimizer. The Source-aware Optimizer extends Catalyst with different categories of rewrite rules. The first category simplifies the *grounded* plan and applies the optimization hints to improve the physical plan quality. The second category maximizes operator pushdown. Finally, the third category examines each data source type in isolation and applies targeted optimizations. We now elaborate on each category.

Rewriting Internal Plan Nodes. After the PV Compiler expands view definitions, the resulting *grounded plan* becomes more complex because additional operations, such as unions and selection predicates, are exposed. The Source-aware Optimizer simplifies this plan by pruning redundant sub-trees and coalescing filtering expressions into disjunctive normal form.

Figure 5.5 presents an optimization instance over the rolling example of Figure 5.1: V_1 is a view that models a union between HDFS and RDBMS-resident data of the *Sales* dataset. Both sides of the union have a filtering predicate applied. When the Source-aware Optimizer examines the filtering predicate of Query $Q1$, it detects that the $Sales$ data in the RDBMS does not need to be accessed to answer the query, and thus rewrites the plan to access only the HDFS-resident data.

After simplifying the plan, if the original analysis task was an SQL query, the Source-aware Optimizer enforces the optimization hints suggested by the SQL Optimizer during Phase I. Specifically, if the SQL optimizer suggests that a join operation must broadcast the smaller dataset involved, the Source-aware Optimizer rewrites the plan to use the appropriate Spark broadcast hash-join operation. The SQL optimizer may also suggest that a sub-tree of the overall query plan must be materialized and then reused later in the same query. In this case, the Source-aware Optimizer injects a $Materialize$ operator in the physical query plan.

Operator Pushdown. When dealing with remote data sources, it is important to reduce the amount of data movement through the network by pushing down operations to them. The

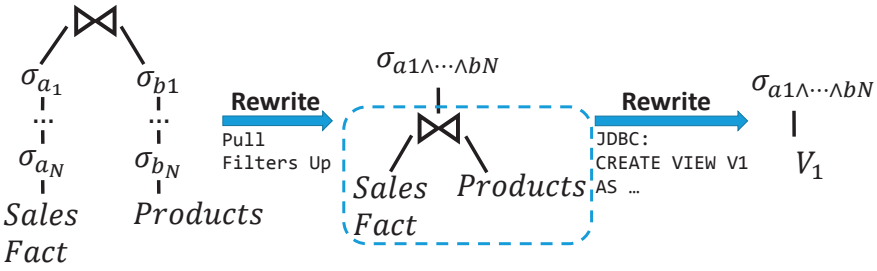


Figure 5.6 – Join pushdown rewriting during source-aware optimization.

Data Source API enables some basic selection and projection pushdown for queries over external sources. However, Catalyst has two major limitations: First, Catalyst is often unable to push down more complex filtering predicates. Second, Catalyst is unable to push down more complex operators such as joins, because the *Data Source API* of Spark SQL is restricted to single-table data accesses. System-PV must thus adress both limitations in a non-intrusive manner, so that it remains compatible with "vanilla" Spark SQL.

First, the Source-aware Optimizer further simplifies filtering predicates to push them down to the external data sources. Second, System-PV performs join pushdown by adding an optimization pass that proceeds as follows: The pass traverses the query plan and finds the largest subtree that contains data accesses to a single data source. If System-PV detects such a subtree, it makes a call to the underlying source to define a temporary view representing the subtree. By exposing the subtree as a single table, System-PV supports join pushdown without harming compatibility with the *Data Sources API* of Spark SQL. Figure 5.6 presents an application of the join pushdown rule over the example of Figure 5.1: Initially, any selection predicates are *pulled above* the join operation, so that the optimization pass has simpler tree patterns to detect. Once a join pattern between two original relations is detected, a temporary view V_1 is created. Finally, selection pushdown is re-applied on the final view, which can be deleted once the query terminates.

Exploiting Source Characteristics. Unlike vanilla Spark, System-PV takes into consideration the characteristics of the different underlying data sources to further optimize the analysis plan. Specifically, the Source-aware Optimizer rewrites queries that are submitted to external data sources in a way that masks the data movement costs.

Large-scale applications pay a significant cost to serialize data, transfer it over the network, and deserialize it [140, 200]. The cost is even more pronounced for Spark when it accesses external data sources: In case of RDBMSs, Spark blindly submits each query through a single JDBC connection; a single Spark task executor is responsible for receiving the data through the network, deserializing it, and shipping results to the other executors to continue query execution. This single task executor often becomes the bottleneck. System-PV, on the other hand, masks data movement cost by rewriting the query into a semantically equivalent union of multiple queries that are concurrently submitted to the RDBMS by multiple Spark task executors. Specifically, the Source-aware Optimizer applies an optimization pass that splits an

RDBMS scan operation into a union of scan operations. The optimization pass is triggered when the data to be scanned i) has an index, or ii) is range-partitioned on the query's predicate(s), which is typical in modern deployments [155]. In these cases, the RDBMS performs selective data accesses, which further improve execution times.

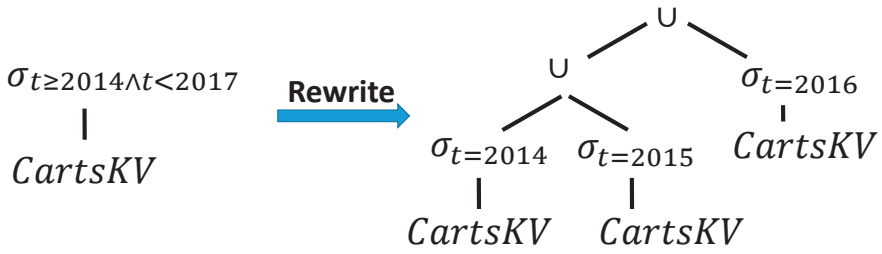


Figure 5.7 – Range query rewriting during source-aware optimization: Data accesses become parallelizable.

System-PV performs a similar optimization when accessing key-value stores, which by design are optimized for queries requesting a single data item by key. The Source-aware Optimizer rewrites range queries on the key attribute into a union of equi-predicate selections to parallelize the ingestion on the Spark side, and also better suit the query capabilities of the key-value store. Figure 5.7 presents an application of said optimization over the example of Figure 5.1: The range predicate is split into a number of equi-predicate selections, and the results of the sub-queries are unified.

Summary. System-PV uses a two-phase optimizer to cover both SQL queries and general analysis tasks, and to reduce the complexity of optimization over multiple data sources. In Phase I, an SQL Optimizer applies SQL-centric optimizations. In Phase II, the Source-aware Optimizer considers the properties of the underlying data sources.

5.7 Experimental Evaluation

We experimentally evaluate System-PV by emulating a business intelligence scenario similar to that of Figure 5.1. The majority of the data is in the data lake (HDFS), whereas the tail end of the data is in a data warehouse (IBM DB2® DPF™) and a key-value store (Cassandra [3]). Our key results are the following:

1. System-PV is faster than Spark SQL over multiple data sources – often by more than an order of magnitude – while masking the complexity of accessing multiple data sources (Section 5.7.2) .
2. The SQL Optimizer of System-PV produces better query plans than Catalyst (Section 5.7.2).
3. The Source-aware Optimizer of System-PV provides significant performance gains by masking the data transfer costs through better parallelization (Section 5.7.2).
4. System-PV accesses the remote data tail end with small overhead added to the case of operating solely on top of the historical data in the data lake (Section 5.7.3).

5.7.1 Experimental Setup

We use the TPCx-BB benchmark [20, 114] data generator at scale factor 1000 to populate the **web_clickstreams** table (180 GB) and **web_sales** table (450 GB). To incorporate non-relational data, we additionally generate the **web_events** dataset (90 GB) that contains sales data that has been produced by mobile devices in JSON format. The **web_clickstreams** table is entirely stored in the data lake to emulate the case in which data is directly ingested in HDFS. The **web_sales** table is split between HDFS and DB2 DPF. This is because information about sales is typically inserted in an RDBMS and periodically loaded in the data lake. Similarly, the semi-structured **web_events** dataset is split between HDFS and Cassandra.

We use Spark version 1.4.0 on a 10 node cluster, DB2 DPF version 10.1.0 on a 5 node cluster, and Cassandra version 2.1.7 on a 4 node cluster. All nodes are equipped with two 6-core Intel Xeon E5-2430 CPU @ 2.20GHz, 96GB RAM, and 11 × 2TB SATA disks. The nodes are connected through a 10 Gbit Ethernet switch.

The experiments compare four *data placement configurations*: In the first case, 90% of the Sales and Events tables reside in HDFS, and the 10% left resides in DB2 and Cassandra, respectively (**90-10**). In the second case – the closest to real-world scenarios – 99% of the Sales and Events tables reside in HDFS, and the 1% left resides in DB2 and Cassandra, respectively (**99-1**). In both cases, data is range-partitioned based on a date attribute. Finally, the third and fourth cases represent baseline extremes: Either all datasets are entirely stored in HDFS (**Local**), or each dataset resides in a different data store (**Remote**). **Local** represents the scenario where the users access only the data in the data lake and thus ignore data freshness.

We use a query template that represents a scenario which is frequent in data lake environments: combining data from all the involved data sources. The template $T(X,Y,Z)$ – shown below – includes a 3-way join and a number of filtering predicates with non-fixed selectivities (X,Y,Z). The template allows us to generate various types of queries that stress different parts of a system. By using different combinations of predicate selectivities, we affect the amount of data to be transferred across sources, and also evaluate the query processing and optimization capabilities of System-PV, given that different selectivities can trigger different join orders.

```
1 SELECT AVG(s_sales_price)
2 FROM web_clickstreams c
3 JOIN web_sales ss ON
4      (c_user_sk = s_bill_customer_sk)
5 JOIN web_events e ON
6      (s_bill_customer_sk = e_cust_id)
7 WHERE (c_click_date_sk BETWEEN X1 AND X2)
8  AND  (s_sold_date_sk BETWEEN Y1 AND Y2)
9  AND  (e_session_date BETWEEN Z1 AND Z2)
```

Listing 5.4 – Query template for analysis across data sources.

5.7.2 System-PV vs. Spark

We compare System-PV with Spark by quantifying the impact of each of the two System-PV optimization phases.

System-PV SQL Optimizer vs. Spark Catalyst

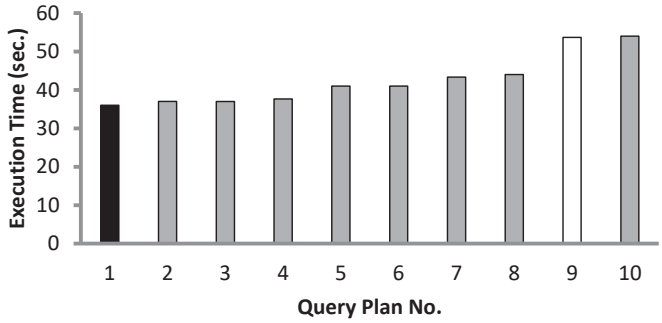
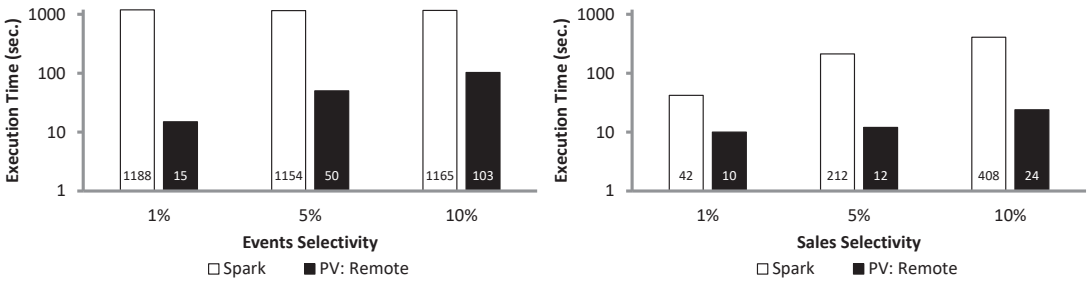


Figure 5.8 – Query Plan Quality: The SQL Optimizer of System-PV picks the best candidate plan (No. 1), whereas Spark's Catalyst optimizer picks plan No. 9.

The goal of this experiment is to validate that the SQL optimizer of System-PV produces efficient plans. We generate the instantiation $T(1,5,10)$ of the query template in Listing 5.4, namely Q, which selects 1% of the Clickstreams data, 5% of the Sales data, and 10% of the Events data using the **90-10** data placement configuration. We compare the query plan generated by the System-PV SQL optimizer for Q (Query Plan No.1 in Figure 5.8) against various other plans in the space of all plans for queries generated from the template of Listing 5.4.

We choose not to pick random plans from the plan space for this comparison because they are highly likely to exhibit dramatically poor performance. Instead, we pick plans that are potentially close to the optimal. One such plan is the one generated by the Catalyst optimizer (Query Plan No.9). Other selected plans were the ones generated by the System-PV SQL optimizer for various other template instantiations. These plans are shown in gray in Figure 5.8. We execute multiple runs of query Q **on System-PV**, each time hand-coding a different virtual plan corresponding to one of the selected plans, and using the same source-aware optimizations for all of them.

Different plans lead to different execution times – a fact that further highlights the need for a cost-based optimizer. System-PV picks plan No.1 ($(Sales \bowtie Events) \bowtie Clicks$), which builds hash tables on the Clickstreams and Events datasets (i.e., the right operands of each join) and probes them using records from the Sales dataset (i.e., the left operand). This plan ends up being the best choice because the Clickstreams dataset is stored in the data lake and thus System PV builds a hash table over each node's local data in parallel. The Catalyst optimizer, on the other hand, picks plan No.9 ($(Clicks \bowtie Sales) \bowtie Events$) – the second worst from the plans tested. Plan No.9 builds hashtables over the Sales and the Events datasets, which it then probes using the records of the Clickstreams dataset. Both Sales and Events, however, have a significant portion of data stored in remote sources and thus require additional effort to



(a) Clicks and Sales Selectivities kept at 0.1%.

(b) Clicks & Events Selectivities kept at 0.1%.

Figure 5.9 – Spark vs. System-PV: Spark is unable to keep up with System-PV even for very selective queries.

build the hashtables. In addition, their corresponding predicates are less selective than the predicate on the Clickstreams dataset.

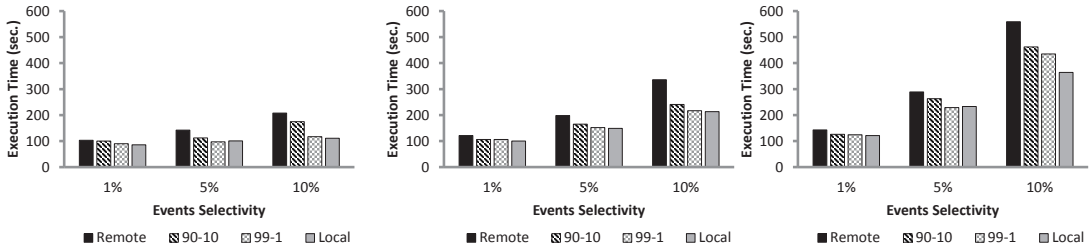
Our results show that unlike Catalyst, the System-PV SQL optimizer considers the predicate selectivities and the location of a dataset over which a hashtable is built to produce an efficient plan. We repeated this analysis using the same protocol but starting with different instantiations of the template T using other selectivity values and data placement configurations as well: We obtained similar results.

Impact of Source-aware Optimizer

We now quantify the performance gains that System-PV has over Spark due to the Source-aware Optimizer. We make sure that both System-PV and Spark use the same optimal *virtual plan* by hand-coding the plan produced by the System-PV SQL optimizer. As shown in Section 5.7.2, in many cases Spark picks a suboptimal plan, and thus the Spark performance results presented here are conservatively optimistic.

We test the **Remote** data placement configuration – the most challenging of the ones examined – by instantiating the template T with different selectivity values for the predicates; we generate 6 queries in total. The predicates touching two of the three datasets are kept very selective. Less selective configurations stressed Spark even more; we omit them in the interest of space. We vary the selectivity of the predicate over the third dataset, so that the amount of data fetched from the remote source varies too.

Figure 5.9a presents the case in which only 0.1% of the HDFS-resident clickstreams and the DB2-resident Sales are selected. The selection predicate for the Cassandra-resident Events ranges from 1% to 10%. In this case, System-PV is 11× to 79× faster than Spark. Note that the execution time of System-PV increases as the query becomes less selective, and more data has to be fetched from Cassandra. Spark, on the other hand, shows little variation in execution time regardless of the amount of data to be fetched. The reason is that Spark attempts to push a range (sub-)query down to Cassandra, which Cassandra is unable to process. Thus, Cassandra ships the entire dataset to Spark through a single-threaded connection, and Spark



(a) Clicks: 10%. Sales: 1%. (b) Clicks: 10%. Sales: 5%. (c) Clicks: 10%. Sales: 10%.

Figure 5.10 – System-PV performance for various data placement configurations and query selectivities.

then applies locally the range predicate. On the contrary, System-PV rewrites the range query into a union of equi-predicate selections that it concurrently submits to Cassandra. These standalone sub-queries are served in parallel, thus resulting in fast data ingestion rates.

Figure 5.9b presents the case in which 0.1% of the HDFS-resident Clickstreams and the Cassandra-resident Events are selected. The selection predicate for the DB2-resident Sales ranges from 1% to 10%. Note that for this experiment, we incorporated the System-PV optimizations targeted for key-value stores into Spark SQL. These optimizations were enabled in both Spark SQL and System-PV in order to quantify the performance benefits attributed to the database-related rewrites of System-PV in isolation. System-PV is again 4× to 17× faster than Spark because it parallelizes data transfer from DB2. Note that Spark SQL *does* successfully pushdown the selection predicate to DB2, but retrieves the data through a single-threaded connection, which ends up being the bottleneck for the entire query.

5.7.3 System-PV Performance

We now evaluate System-PV using all four data placement configurations and varying the amount of fresh data transferred over the network. Our aim is to verify that System-PV performance for split data scenarios is comparable to the scenario of solely operating on top of the historical, stale data. We exclude Spark from this discussion because its running time always exceeded 1000 seconds; as Section 5.7.2 showed, Spark is unable to keep up with System-PV even for selective queries that require small network data transfers. The results show that System-PV uses the optimizations of Section 5.6.2 to mask the cost of remote data accesses, and thus provides similar performance to solely operating on top of the data lake.

Figure 5.10 presents 9 instantiations of the query template. All of them select 10% of the HDFS-resident Clickstreams. The queries select either 1% of the Sales dataset (Figure 5.10a), 5% (Figure 5.10b), or 10% (Figure 5.10c). We vary selectivity over the Events dataset in every query to gauge the effect of accessing the slowest data source (i.e., Cassandra).

As seen in Figure 5.10a, all the data placement configurations have similar performance, with **Remote** being slightly slower than the others. The performance gap opens in the case of 10% selectivity; even then, however, the performance observed with the **99-1** configuration is al-

most identical to the best-case scenario where remote data access does not occur (**Local**). The reason is that the source-aware optimizations mask the remote data access cost by overlapping data transfer with query execution.

The queries shown in Figure 5.10b are more expensive than the ones of Figure 5.10a because a bigger subset of the Sales dataset participates in the join. Still, the **90-10** and **99-1** configurations exhibit execution times similar to the **Local** configuration. Even in the case of 10% selectivity, the execution times corresponding to the **99-1** and **Local** configurations are only 4 seconds apart, thus denoting that System-PV is again able to mask remote data accesses.

Figure 5.10c presents the least selective version of the experiment. When selectivity over the Events dataset is 1%, all data placement configurations except **Remote** have almost identical execution time. Note that although the sub-query pushed down to DB2 is non-selective, System-PV splits and parallelizes the sub-query, thus hiding the increased data transfer cost. When selectivity over the Events dataset reaches 10%, the gap between **Remote** and **Local** increases. Note, however, that the **Local** configuration misses the latest fresh data. The performance difference stems from the simultaneous increase of i) remote data accesses, ii) the amount of data shuffled due to the distributed hash join, and iii) the size of intermediate results, all of which stress the network bandwidth. Finally, the performance of System-PV in the common split-dataset configurations (**99-1**, **90-10**) is similar to that observed when accessing all the data locally (**Local**).

Summary. System-PV significantly outperforms Spark, even for the worst-performant scenario of accessing federated data sources (**Remote**). In addition, when testing System-PV under different data placement configurations, the response times for the two extreme cases (fully local vs. fully remote) start to diverge; still, for the split-dataset cases that System-PV targets, response times are comparable to that of the best-performant, fully local scenario (**Local**), without compromising data freshness.

5.8 Perspectives

Our experience with Spark and other similar frameworks has shown us that although they support various types of data analysis over historical data in a data lake, they lack the necessary abstractions to query data sets spread across multiple data sources, thus rendering the overall analysis complex for the user. At the same time, their performance is suboptimal when accessing external sources.

System-PV introduces a high-level abstraction in the form of a global virtual schema, which hides source complexity from users and allows them to seamlessly access both the historical as well as the latest data. System-PV also optimizes both SQL and procedural analysis tasks through a unique two-phase query optimization approach. System-PV thus supports a broad spectrum of data usage patterns: an individual dataset can be accessed in the remote source completely, can be split between the data lake and the remote source, or can be accessed locally in the lake (if the application can tolerate data staleness). In addition, splits of a dataset

can overlap; System-PV chooses from which source to retrieve the overlapping part depending on the user's data freshness requirements.

Using System-PV in practice has led us to a number of observations that allow reaching its full potential, and that can be useful as guidelines to system designers working on split-data scenarios.

5.8.1 System-PV for enterprise workloads

Enteprise data management architectures typically model data using a variation of the star or the snowflake schema, which involve few large fact tables and numerous smaller dimension tables [151].

Small datasets. System-PV masks the cost of accessing remote datasets of small size, such as dimension tables of a star schema. Given that dimension tables receive frequent updates, and that different parts of an organization often join their own versions of dimension tables against a fact table [151], we propose storing dimension tables only in the original, external data sources; there is no need to store them in the data lake as well, since accessing them with System-PV has minimal overhead.

Fact tables. Large fact tables receive append-like updates, and users typically set up an ETL process to archive the data appends in the data lake. System-PV by default accesses both the local and the remote part of a fact table. If possible, we suggest running ETL frequently, so that running analysis with System-PV over both parts of the fact table has comparable performance to accessing only the local part. In addition, more data accumulates in the lake over time, whereas the size of the remote delta remains stable, thus the cost of remote accesses appears small due to the order-of-magnitude difference in local and remote data sizes.

Minimizing data transfers. The source-aware optimizations of System-PV that generate sub-queries to parallelize external data retrieval provide their maximum benefit when the external sources offer a way to reduce the amount of data that each sub-query accesses. For key-value stores, a query on the key of each object naturally accesses a small amount of data. For RDBMS, populating indexes on fields that are popular query predicates, or partitioning the data, is helpful. Given that primary and foreign keys are typically coupled with indexes, enterprise star and snowflake schemata already have useful indexes in place. Therefore, System-PV applies its rewriting optimizations without requiring an additional indexing storage overhead.

5.8.2 Optimizing SQL-on-Hadoop performance over multiple sources

Apart from the user-friendly virtual schema that System-PV employs, it also makes use of multiple performance optimizations that improve the performance of Spark scripts over dispersed datasets. It is worth examining whether these optimizations can also be applied to existing systems even if said systems currently lack first-class support for data virtualization.

There is a number of ways in which existing SQL-on-Hadoop systems can be adjusted to improve their performance over diverse data stores. We use Spark SQL as an example and consider its architecture in a top-down fashion.

Starting from the query optimizer, Catalyst is a significant effort towards performing optimization across multiple types of analysis. However, it is currently not as mature as several traditional, specialized database optimizers that have been refined over multiple years. Thus, we believe that Catalyst must also introduce interfaces that allow users to "plug" their optimizer of choice based on the type of analysis they intend to launch[3]. Users can choose among optimizer modules, such as the one of System-PV, Orca [230], Calcite [2], etc.

Our experience building System-PV showed us that integrating the Source Optimizer's rewrite rules into Catalyst is straightforward and would be a valuable addition to Spark. Still, applying the source-aware rewrites of System-PV requires examining carefully the properties of the underlying systems, and triggering the rewrites judiciously. For example, triggering the query rewrite for range predicates that access non-key fields in a key-value store, or for arbitrary, non-partitioning / non-indexed fields in a DBMS table, can significantly penalize performance. Therefore, Spark must be able to acquire and store information/statistics from the underlying data stores to make educated rewriting decisions.

Instead of applying some of the source-aware optimizations in Catalyst, one could extend/rewrite the data connectors of Spark to reduce the cost of accessing and transferring remote data into the data lake. As shown by this work, one way to reduce the cost is by parallelizing the subquery that accesses a remote store. In addition, data connectors can perform data exchange using a portable, binary wire format such as Arrow [21]; Arrow has the same in-memory and on-wire representation, and thus reduces the effort spent in data (de)serialization, which is a major cost in data-center-scale analytics [200, 140].

Summary. System-PV provides a spectrum of choices for data freshness and where to access the data in complex enterprise data ecosystems. Combined with the guidelines above, System-PV forms a comprehensive solution for ad-hoc data analysis in enterprise settings, which can also influence the design of state-of-the-art SQL-on-Hadoop systems.

5.9 Summary

We present System-PV, a system that supports various types of analysis over multiple data sources. System-PV addresses the shortcomings of the state-of-the-art systems by extending Spark with a data virtualization module that masks data source complexity. It uses a location-aware compiler and a sophisticated two-phase optimizer to optimize user scripts over a global virtual schema. Our results show that System-PV is significantly faster than Spark when querying dispersed datasets, and introduces small overhead for accessing the remote tail-end of the data compared to operating solely on top of the data lake.

[3] Spark appears to be already moving in this direction [214].

6 Unified Scale-Out Data Cleaning

Data cleaning is an indispensable part of data analysis due to the increasing amounts of dirty data. Data analysts spend most of their time preparing dirty data before it can be used for analysis. At the same time, the existing tools that attempt to automate the cleaning procedure typically focus on a specific use case and operation. Still, even such specialized tools exhibit long running times or fail to process large datasets. Therefore, from a user's perspective, one is forced to use a different, potentially inefficient tool for each category of errors.

This chapter addresses the coverage and efficiency problems of data cleaning. It introduces CleanM (pronounced *clean'em*), a language that can express multiple types of cleaning operations. CleanM goes through a three-level translation process for optimization purposes; a different family of optimizations is applied in each abstraction level. Thus, CleanM can express complex data cleaning tasks, optimize them in a unified way, and deploy them in a scaleout fashion. We validate the applicability of CleanM by using it on top of CleanDB, a newly designed and implemented framework that can query heterogeneous data.

6.1 Introduction

Today's ever-increasing rate of data volume and variety opens multiple opportunities; crawling through large-scale datasets and analyzing them together reveals data patterns and actionable insights to data analysts. However, the process of gathering, storing, and integrating diverse datasets introduces several inaccuracies in the data: Analysts spend 50%-80% of their time preparing dirty data before it can be used for information extraction [172]. Therefore, data cleaning is a major hurdle for data analysis.

Data cleaning is challenging because errors arise in different forms: Syntactic errors involve violations such as values out of domain or range. Semantic errors are also frequent in non-curated datasets; they involve values that are seemingly correct, e.g., Beijing is located in the US. In addition, the presence of duplicate entries is a typical issue when integrating multiple data sources. Besides requiring accurate error detection and repair, the aforementioned

data cleaning tasks also involve computationally intensive operations, such as inequality joins, similarity joins, and multiple scans of each involved dataset. Thus, it is difficult to build general-purpose tools that can capture the majority of error types and at the same time perform data cleaning in a scalable manner.

Existing data cleaning approaches can be classified into two main categories: The first category includes interactive tools through which a user specifies constraints for the columns of a tabular dataset or provides example transformations [139, 212]. User involvement in the cleaning process is intuitive and interactive, yet specifying all possible errors involves significant manual effort, especially if a dataset contains a large number of discrepancies. The second category comprises semi-automatic tools that enable several data cleaning operations [90, 110, 149, 235]. Both categories lack a universal representation for users to express different cleaning scripts, and/or are unable to optimize different cleaning operations as one unified task because they treat each operation as a black-box UDF.

Therefore, there is need for a higher-level representation for data cleaning that serves a purpose similar to that of SQL for data management in terms of expressivity and optimization: First, SQL allows users to manage data in an organized way and is subjective to how each user wants to manipulate the data. Similarly, data cleaning is a task that is subjective to the user's perception of cleanliness and therefore requires a language that allows users to express their requests in a simple yet efficient way. Second, SQL is backed by the highly optimizable relational calculus; data cleaning tasks require an optimizable underlying representation too.

This chapter introduces CleanM, a declarative query language for expressing data cleaning tasks. Based on SQL, CleanM offers primitives for all popular cleaning operations and can be extended to express more operations in a straightforward way. CleanM follows a three-level optimization process; each level uses a different abstraction to better suit the optimizations to be applied. First, all cleaning tasks expressed using CleanM are translated to the *monoid comprehension calculus* [105]. The monoid calculus is an optimizable calculus that is inherently parallelizable and can also represent complex operations between various data collection types. Then, comprehensions are translated into an intermediate algebra, which allows for inter-operator optimizations and detection of work sharing opportunities. Finally, the algebraic operators are translated into a physical plan, which is then optimized for factors such as data skew. In summary, regardless of how complex a cleaning task is, whether it internally invokes complex operations such as clustering, and what the underlying data representation is (relational, JSON, etc.), the overall task will be treated as a single query, optimized as a whole, and executed in a distributed, scale-out fashion.

We validate CleanM by building CleanDB, a distributed data cleaning framework. CleanDB couples Spark with a CleanM frontend and with a cleaning-oriented optimizer, which applies the three-level optimization process described above. The end result is a system that combines data cleaning and querying, all while relying on optimizer rewrites and abundant parallelism to speed up execution.

Motivating Example. Consider a dataset comprising customer information. Suppose that a user wants to validate customer names based on a dictionary, check for duplicate entries, and also check whether a functional dependency holds. We will be using this compound cleaning task to reflect the capabilities of CleanM and CleanDB: For example, CleanM enables name validation via token filtering [138] – a common clustering-based data cleaning operation – by representing it as a monoid. Also, CleanDB identifies a rewriting opportunity to merge the duplicate elimination and functional dependency checks in one step.

Contributions: Our contributions are as follows:

- We introduce CleanM, an all-purpose data cleaning query language. CleanM models both straightforward cleaning operations, such as syntactic checks, as well as complex cleaning building blocks, such as clustering algorithms, all while being naturally extensible and parallelizable. We also present a three-level optimization process that ensures that a query expressed in CleanM results in an efficient distributed query plan.
- We implement CleanDB, a scale-out data cleaning framework that serves as a testbed for users to try CleanM. CleanDB supports a multitude of data cleaning operations (e.g., duplicate elimination, denial constraint checks, term validation) over multiple different types of data sources (e.g., binary, CSV, JSON, XML data), executed in a distributed fashion using the Spark platform.
- We show that CleanDB outperforms state-of-the-art solutions in synthetic and real-world workloads. CleanDB scales better than Spark SQL [43] and a dedicated scale-out data cleaning solution, offers a wider variety of operations, and cleans datasets that its competitors are unable to process due to performance issues.

In summary, current data cleaning technology lacks a universal representation that is general and also guarantees scalability out-of-the-box for all the cleaning operations it supports. This chapter provides a solution through an algebraic abstraction, which allows rich features to be embedded in a declarative, optimizable, and parallelizable language. The user can thus intertwine analytics and cleaning using a unified interface over a scale-out system.

6.2 A unified representation

Data cleaning is a computationally intensive process that typically involves multiple iterations over the same dataset and numerous pairwise comparisons of the data records. In fact, many data cleaning tasks would benefit from machine learning operations, such as clustering, to split a dataset into manageable subsets and minimize the number of required pairwise comparisons. Therefore, a data cleaning language must be coupled with a calculus that can support and optimize such operations. At the same time, said calculus must be able to reason about multiple cleaning operations as a whole, and identify inter- and intra-operation optimizations. Besides involving complex operations, data cleaning tasks are typically applied over a variety of data sources and formats. Data that requires curation may be i) relational or not, ii) stored in a DBMS or kept in files, etc. Therefore, a data cleaning language and calculus

must be able to handle data heterogeneity. Finally, given the ever-increasing data volumes, explicit support of parallelism is a prerequisite. This section presents i) the cleaning operations that CleanM supports, and ii) the rationale behind a three-level translation of said cleaning operations into executable code.

6.2.1 Data cleaning operations

In the following, we revisit the data cleaning operations of Section 2.5 to discuss what is required to optimize each operation.

Denial Constraints (DC). DC checks involve a selection or a self-join that detects tuples, pairs of tuples, or groups of tuples that violate the rule. Self-joins are expensive because they involve multiple traversals of the input. Also, as DCs contain arbitrary predicates, such as inequalities, *theta*-joins might be required. Finally, for rules that need to handle non-exact matches, and thus similarity joins may also be required. Similarity joins are costly operations because they involve multiple passes over a dataset, as well as a computationally expensive similarity check per candidate pair.

Duplicate Elimination. Similar to a subset of denial constraints, deduplication involves a similarity self-join to identify potentially duplicate records [134].

Transformations & Term Validation. Semantic transformations involve an *equi*-join or a similarity join with auxiliary data. Specifically, term validation requires the discovery of the most similar words from the dictionary for each word of the dataset. Thus, term validation relies on the efficient computation of similarity checks.

Summary. Efficient handling of self-, theta-, and similarity joins can accelerate multiple cleaning tasks. Besides accelerating standalone operations, having a unified representation for all operations can help in detecting common patterns and work sharing opportunities. Finally, having a principled way to simplify an arbitrary data cleaning script (e.g., unnest nested sub-tasks) makes detection of optimization opportunities over the script more straightforward.

6.2.2 From data cleaning operations to code

This work uses three different abstraction levels to reason about and optimize data cleaning tasks. In the first level, CleanM maps data cleaning operations to the monoid comprehension calculus. As a result, the operations are first-class citizens of the language instead of black-box UDFs. Such composability means that operations can be explicitly used and stacked with each other in monoid comprehensions. Transforming the input dataset between different types and manipulating multiple data types is also possible, a feature exploited by engines that access raw data [143, 144]. Monoid comprehensions are inherently parallelizable and lend themselves perfectly to scale-out execution – a fact that has led existing scale-out approaches to adapt monoids as a core abstraction for data aggregation and incremental query processing [60, 103]. Section 6.3 elaborates on how cleaning operations are mapped to CleanM.

The second abstraction level involves lowering a comprehension into an algebraic form [105], the *nested relational algebra*. Nested relational algebra operators resemble relational operators and are amenable to relational-like optimizations, yet they also explicitly handle complex data types and queries. For example, a user can issue a query combining relational and hierarchical data, and rely on the algebraic translation process to simplify the physical query plan and remove all forms of query nesting. In addition, the algebraic form enables inter-operator rewrites, which coalesce different cleaning operations into a single one and thus reduce the overall cost. Section 6.4 discusses the algebraic rewrites.

The final level specializes the algebraic expression to the underlying execution engine. CleanM currently assumes that Spark [260] is the underlying engine; still, it is pluggable to any scale-out system. This physical level focuses on the particularities of cleaning operations, such as the presence of expensive theta joins. Also, the physical level addresses the absence of uniform distribution in the values of real-world datasets – a fact that can cause load imbalance during data cleaning. Section 6.5 discusses how to generate physical plans that consider both these complications.

6.3 Cleaning data using monoids

CleanM supports multiple cleaning operations, which it internally maps to monoid comprehensions. Still, although a unified representation is important for user convenience, it is also important to optimize each of the operations. In addition, despite the elegance of comprehensions, the goal of CleanM is to serve as a SQL-like higher-level representation that masks the comprehension syntax, given that most users are more familiar with SQL. The syntax of CleanM extends SQL with constructs that express data cleaning operations and handle non-relational data types such as hierarchies; this work focuses on the data cleaning operations. This section presents i) the optimizations that monoid comprehensions allow, ii) the expressive power of CleanM and how to map the building blocks of data cleaning operations to monoids, and iii) the syntax and semantics of CleanM .

6.3.1 Optimizations at the monoid level

CleanM follows a layered design approach. Even in its topmost layer, CleanM distinguishes between high- and low-level operations, both of which are first-class citizens and are expressed using comprehensions. The separation aims at user convenience: High-level operations, such as denial constraints, map directly to a SQL-like, syntactic sugar representation. Low-level operations are internal building blocks for the high-level ones and address the optimization requirements of Section 6.2.1. Both high- and low-level operations go through a rewrite process that applies general-purpose, domain-agnostic optimizations [105].

Domain-agnostic optimizations: Normalization

Regardless of the processing that a comprehension performs, a normalization algorithm [105] puts it into a "canonical" form.

Normalization applies a series of optimization rewrites. Specifically, it applies filter pushdown and operator fusion. In addition, it flattens multiple types of nested comprehensions [150]. It also replaces any function call that appears in a comprehension, with the call's result (*beta reduction*); a function's input can be an arbitrary expression (e.g., a constant, a generator's variable, etc.). In the case of UDFs that are defined as comprehensions themselves, the rewrite results in their unnesting, and facilitates optimizing the rewritten comprehension as a whole. Similar to the SQL-based rewriting of EXISTS clause, normalization unnests existential quantifications. Finally, normalization simplifies expressions that are statically known to evaluate to true/false or to empty collections.

The result of the normalization process is a simplified comprehension; Section 6.4 explains how this comprehension is further rewritten into a form more suitable for efficient execution.

Domain-specific optimizations: Pruning comparisons

Besides domain-agnostic optimizations, the monoid calculus can express operations that specifically target and accelerate data cleaning tasks. A common theme of all the data cleaning operations mentioned in Section 6.2.1 is the need for fast pairwise comparisons. The rest of this section discusses how to optimize CleanM expressions on the comprehension level by pruning comparisons in the cases of self-joins and similarity joins; we discuss the rest of the optimization requirements of Section 6.2.1 in subsequent sections because they are a better match for lower abstraction levels.

Self-joins occur in denial constraints (DC) and duplicate elimination. In the case of self-joins that involve equality conditions, such as in functional dependencies (FD), CleanM avoids the self-join by grouping the dataset's entries based on the left hand side of the FD, and then detects violations (i.e., whether a grouping key is associated with more than one value). Section 6.5 discusses how CleanM handles the general case of DCs, which may involve non-equality predicates, in its third abstraction level – the physical one.

Regarding similarity joins, a baseline method to evaluate them would compute the cartesian product and afterwards apply a filter that removes the dissimilar pairs. The baseline approach, however, is very costly, because both the cartesian product and the string similarity computation are expensive tasks. Thus, CleanM uses a filtering phase to prune the candidate pairs that need to be checked. An indicative example of filtering is the use of a clustering algorithm to create k clusters, each containing words that are similar. Then, the cleaning operation only has to perform intra-cluster comparisons. The pre-processing filtering phase must be lightweight enough to avoid adding an overhead that reaches the cost of an unoptimized implementation. Thus, CleanM considers variations of the approaches suggested in [138, 220], namely k-means

and token filtering, because different clustering/filtering techniques are more suitable for different use cases; their efficiency in the context of data cleaning depends on several factors, such as the string length of a dataset's words and the similarity metric used. Still, to use any technique, we must be able to express it as a monoid.

6.3.2 Expressive Power: Mapping cleaning building blocks to the monoid calculus

Expressing an operation over type T as a monoid involves either mapping the operation to an existing monoid or proving three properties: First, specifying an identity/zero element $\mathbb{Z}_{\oplus}$ such that for any element of type T, $x + \mathbb{Z}_{\oplus} = \mathbb{Z}_{\oplus} + x = x$. Second, specifying a unit function that turns an element into a singleton value of T. Third, showing that the associative property $\oplus$ holds for it. Multiple operations over collections such as lists, bags, sets, arrays, vectors, etc., are provably mappable to the monoid calculus [105]. Also, monoid comprehensions are sufficient to represent OQL and SQL queries [105]. The rest of this section elaborates on how to map clustering and filtering algorithms – which CleanM relies on to refine similarity joins – to the monoid calculus.

Clustering as a monoid

Clustering algorithms can be divided into partitional and hierarchical. Below, we map each category to the monoid calculus.

Single-pass partitional algorithms. Partitional algorithms split the input into a number of clusters. Each element of the dataset might belong to exactly one (strict) or more clusters (overlapping). The assignment of a value to a cluster depends on certain criteria, such as the distance from the cluster center (k-means) or the distance from the other elements of the cluster (DBSCAN). In the following, we provide the mapping of k-means – the most popular partitional algorithm – to the monoid calculus; mapping other partitional algorithms to the monoid calculus is straightforward by mapping different cluster assignment criteria.

K-means assigns each input element to the cluster that contains values that are similar to it; thus, when used in the context of similarity joins, only intra-cluster comparisons take place. CleanM by default uses a variation of k-means inspired by ClusterJoin [220]. The k-means variation selects k random centers and then assigns each word of the dataset to all centers whose distance is minimum (or minimum plus a $delta$ to favor multiple assignments). The original k-means requires multiple iterations before converging to an optimal set of clusters, which hurts scalability. The k-means variation avoids scalability issues by only iterating once over the input, while also achieving a "good-enough" grouping of similar words.

Mapping the k-means single-pass operation over bag collections to the monoid calculus requires expressing the *center initialization* and the *center assignment* steps as monoid operations; the latter step is the one performing the actual clustering/partitioning.

We express *center initialization* by parameterizing *the function composition monoid* [105] instead of defining a new monoid. The function composition monoid can compose functions that propagate a state during an iteration over a collection, as long as the composed functions are associative. The "propagated state" at the end of the iteration comprises the centers for k-means.

We can parameterize the function composition monoid to apply randomized algorithms, such as reservoir sampling [247], to extract k centers. A possible parameterization is the following:

$$\circ\{\lambda(x,i).(if\ i in N/k, 2N/k, ..., N, then [x]++y, i-1) | y \leftarrow Y\}.$$

The formula iterates through collection Y. The state that the formula propagates in each step of the iteration is the value for i, which initially corresponds to the length of the input collection, and is decreased by 1 in each step. For every element visited, the formula checks whether the element's index in the input collection is $N/k, 2N/k, ...$, or N. If so, it appends the current element y to the output list of centers. Extracting items using a fixed step is an associative operation because it appends specific elements to a collection per iteration, thus the overall parameterization of the composition monoid is a monoid operation too.

Center assignment takes as a parameter the list of centers computed in the first step and discovers the closest center for each data item. This operation maps to the $Minimum$ monoid [105].

Multi-pass partitional algorithms. Representing multi-pass partitional algorithms (e.g., the original k-means, canopy clustering [179], etc.) as monoids is straightforward: The representation of iterative clustering algorithms implies n equivalent monoid comprehensions, where n is the number of iterations. Each iteration stores the result of the comprehension into a state which is then transferred to the next iteration. Alternatively, an *iteration monoid* can act as syntactic sugar in place of the n comprehensions; its behavior will resemble *foldLeft*, and it will update some state in each iteration.

Hierarchical clustering. Hierarchical clustering generates clusters that can have sub-clusters. Executing hierarchical clustering involves a set of iterations that gradually build the resulting clusters by merging or splitting items. In the monoid representation of hierarchical clustering, each iteration gets as input the previous state or the initial dataset, and computes the items whose distance from each other is minimum; this operation maps to the Min monoid.

(Token) filtering as a monoid

Token filtering [138] is the preferred way to reduce the number of comparisons in similarity joins when comparing strings of small length, whereas clustering-based filtering is suitable for more generic use cases. The algorithm groups the words based on their tokens in order to avoid comparing all pairs exhaustively. Specifically, token filtering splits each word into tokens of length q, and then associates each token with the groups of words that contain the same token. Therefore, similarity checks only take place within each group.

The monoid representation of token filtering resembles that of k-means, in that k-means

groups values based on their common "center", whereas token filtering groups them based on a common token. Below, we provide the mapping of token filtering into the monoid calculus. $[str_i, str_j, str_k]$ denotes that at least one of the three strings will be part of the set of values that contain the token.

$$\mathbb{Z}_{\oplus} : \{\}, \; Unit : str \rightarrow \{(token_i, \{str\}), (token_j, \{str\})...\}$$
$$Associative\ property : tokenize(str_i, tokenize(str_j, str_k)) =$$
$$\{(token_i, \{[str_i, str_j, str_k]\}), (token_j, \{[str_i, str_j, str_k]\})...\} =$$
$$tokenize(tokenize(str_i, str_j), str_k)$$

Extensibility and scope of CleanM

Extending CleanM with any operation that obeys the monoid properties is straightforward. Besides k-means clustering and token filtering, CleanM can represent any filtering approach that groups words into clusters of similar contents (e.g., filtering based on the length of the words). Other filtering approaches, such as applying transitive closure to build the similar pairs, can be also represented using the monoid calculus.

Future work includes examining operations which lack an associative property (e.g., median), and which have traditionally been handled by scale-out systems via exponential algorithms or approximation. Finally, this work focuses on violation detection with minimal user effort; cleaning-oriented topics such as i) data repairing techniques and ii) techniques that rely on classification using an offline training phase and pre-existing training data are orthogonal extensions to our declarative language proposal.

6.3.3 The CleanM language

Having defined the necessary low-level operations, we describe the high-level cleaning operations of CleanM. CleanM extends SQL with data cleaning operators; its syntax is shown in Listing 1. The symbols ([]), (*) and (|) denote optional elements, elements that can appear multiple times, and choice between elements, respectively. The symbol (|) implies arbitrary order between the options. When multiple cleaning operations appear in the CleanM query, then the semantics of the query correspond to an outer join that takes as input the violations of each cleaning operator that appears in the query and outputs the entities that contain at least one violation. Except for the `[FD|DEDUP|CLUSTER BY]` part, the syntax and semantics of the operators are equivalent to that of SQL.

```
SELECT [ALL|DISTINCT] <SELECTLIST>
<FROMCLAUSE>
[WHERECLAUSE][GBCLAUSE[HCLAUSE]][FD|DEDUP|CLUSTER BY]*

FD=FD(attributesLHS, attributesRHS)
DEDUP=DEDUP(<op>[,<metric>, <theta>][,<attributes>])
CLUSTERBY=CLUSTER BY(<op>[,<metric>,<theta>],<term>)
```

Listing 1 – The syntax of CleanM. CleanM extends SQL with the operators FD, DEDUP, and CLUSTERBY

We now analyze the syntax of each operator and present the semantics of CleanM using the monoid calculus. We also go through the running example of the introduction, which checks the rule $address \rightarrow prefix(phone)$, detects duplicate customers, and validates customer names using token filtering and a dictionary. The corresponding CleanM query is the following:

```
SELECT c.name,c.address, *
FROM customer c, dictionary d
CLUSTER BY(token filtering, LD, 0.8, c.name)
FD(c.address, prefix(c.phone))
DEDUP(token filtering, LD, 0.8, c.address)
```

Denial Constraints. The general category of denial constraints is expressible using vanilla SQL, thus CleanM reuses SQL syntax to express them. CleanM makes an exception for functional dependencies – the most popular sub-category of denial constraints – and uses the FD operator shown in Listing 1. The query result contains the entities that violate the FD rule. *LHS* and *RHS* correspond to the left and right-hand side of the rule. Both *LHS* and *RHS* can involve more than one attribute. The semantics of the FD operator correspond to the following comprehension:

```
groups:=for(d<-data) yield filter(d.term,algo),
for(g<-groups,g.count>1) yield bag g
```

The comprehension groups the input dataset using the *filter* monoid based on a *term* attribute to reduce the pairwise comparisons required and returns the groups containing more than one item. The filter monoid is a placeholder for kmeansFilter, tokenFilter, or a plain groupBy that behaves like its SQL counterpart.

The functional dependency rule $address \rightarrow prefix(phone)$ of the example corresponds to the following comprehension:

```
groups:=for(c<-cust) yield groupBy(prefix(c.phone)),
for(g<-groups,g.count>1) yield bag g
```

Duplicate Elimination. The DEDUP operator of Listing 1 comprises the <op> field that represents the filter to use for the similarity join, <metric>, which is the distance metric to be used (e.g., Jaccard, Euclidean), and <theta> – the similarity threshold. The <attributes> field represents the set of attributes that determine whether two entities are equal. <attributes>, <metric> and <theta> are optional – a default value is set if they are missing. The query result contains the duplicate entities. The semantics of the DEDUP operator correspond to the following comprehension:

```
groups := for(d <- data) yield filter(d.terms,algo),
for(g<-groups,p1<-g.partition,p2<-g.partition,
    similar(metric,p1.atts,p2.atts,θ))
yield bag(p1, p2)
```

The *filter* monoid groups the data based on the specified attributes or by building clusters based on that attributes. Then, the entries within each group are compared against each other using a similarity metric. The comprehension outputs pairs that are potential duplicates. `partition` is a built-in field that represents the set of records that correspond to each group. `LD` is a shortcut for the Levenshtein distance (LD) similarity metric. The comprehension of the deduplication part of the running example is the following:

```
groups:=for(c<-cust) yield filter(c.address,tf),
for(g<-groups,p1<-g.partition,p2<-g.partition),
LD(p1.atts,p2.atts)>0.8) yield bag(p1, p2)
```

Term Validation. The CleanM syntax for term validation requires the `CLUSTER BY` operator of Listing 1, which resembles `DEDUP`. The <term> field stands for the attribute(s) based on which the similarity is measured. `CLUSTER BY` requires also an additional table in the <`FROMCLAUSE`> that represents the dictionary.

The query result couples each dirty term with the set of dictionary terms that are similar to it. The similar dictionary terms correspond to the suggested repair of the invalid term. The semantics of `CLUSTER BY` correspond to the following comprehension:

```
dataGroup := for(d<-data) yield filter(d.term,algo),
dictGroup := for(d<-dict) yield filter(d.term,algo),
similarTerms := for(d1<-dataGroup, d2<-dictGroup,
                    d1.key = d2.key,
                   similar(metric,d1.term,d2.term,θ))
                yield list(d1.term, d2.term)
```

First, the input is clustered based on a *term* attribute whose values potentially contain inconsistencies. The same process is followed for the entries of the dictionary. Then, the comprehension tries to find similar data-dictionary pairs by comparing only the clusters that correspond to the same grouping key. The respective validation of the customer name in the running example is the following:

```
dataGroup := for(c<-cust) yield filter(c.name,tf),
dictGroup := for(d<-dict) yield filter(d.name,tf),
similarTerms := for(d1<-dataGroup, d2<-dictGroup,
d1.key = d2.key,LD(d1.name,d2.name)>0.8)
yield list(d1.name, d2.name)
```

Transformations. CleanM differentiates between syntactic and semantic transformations. Syntactic transformations are lightweight repair operations, such as splitting an attribute, and thus can be expressed using vanilla SQL. Semantic transformations require an auxiliary table that contains value mappings. Thus, they reuse the term validation constructs, with the

difference that the projection list contains the desirable attribute from the auxiliary table as a suggested repair. For example, one could map airports to cities using an auxiliary table that contains airport-to-city mappings.

Summary. CleanM exposes users to a SQL-like extension: Each operator extends the syntax of SQL based on the functionality it resembles. Every operator is deeply integrated in CleanM instead of being treated as a black-box UDF; all operators end up translated to the monoid comprehension calculus. Thus, CleanM treats cleaning operations as inherently parallelizable, offers operation composability, and can operate over non-relational data. The monoid representation allows for high-level optimizations, influenced by data mining techniques, that avoid the computation of cross products during data cleaning. The next two sections present representations that are more suitable for additional optimization tasks.

6.4 Unified algebraic optimization

The optimizations at the monoid comprehension abstraction level result in a rewritten comprehension. While the comprehension has undergone optimizations such as filter pushdown and partial unnesting, there are still opportunities for optimizing the overall cleaning task. Therefore, as described in Section 2.2, the second abstraction level translates a comprehension into a *nested relational algebra* expression [105], which is more suitable for the next round of CleanM optimizations. The full algorithm for rewriting a comprehension to an algebraic plan is presented in [105]; the result is a logical plan that uses the operators of Table 2.2.

There are three major benefits from the algebraic representation: First, there exist rules, that remove any leftover query nestings [105]. Unnesting simplifications is useful in data cleaning, since query and data nestings are inherent in cleaning operations. Second, by expressing all different monoid types into a common, confined algebra, it becomes possible to detect opportunities for intra-operator and inter-operator optimizations, such as work sharing between operators. The running example depicted in Figure 6.1 shows the first two benefits. Finally, by translating comprehensions into an algebraic form, the optimization techniques that have been proposed in the context of the established relational algebra become applicable over an unnested, simplified query representation.

Optimizations at the algebra level

CleanM queries benefit from many expression simplifications that are possible at query rewrite time [105]. After having removed the nestings of the query, apart from the relational algebra optimizations, the optimizer can detect common patterns and enable work sharing between operators. In the following we present the simplifications that the query of the running example goes through.

The query checks for invalid terms, duplicates, and functional dependency violations. A baseline approach would treat each cleaning operation as a separate task that traverses the

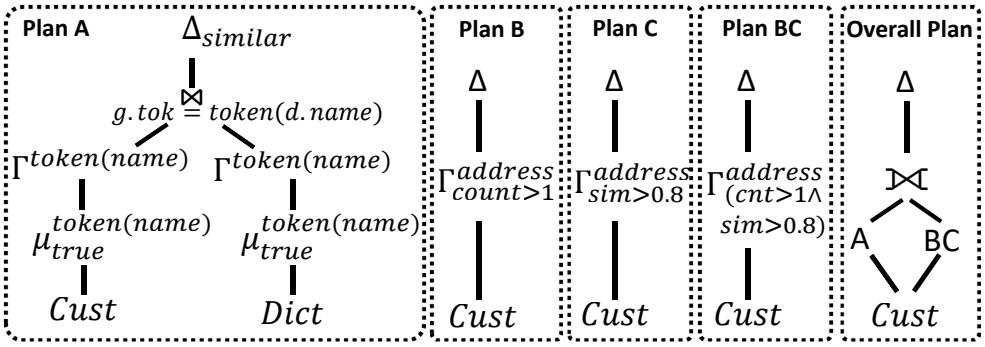


Figure 6.1 – Algebraic plans for our running example, and optimized rewritten plans that coalesce operators and share work.

input and detects violations. Treating each operation on its own results in the plans A, B, C of Figure 6.1. Plan A performs term validation via token filtering: It unnests the list of names in order to compute the tokens of each name, then groups by token to detect similar names. By injecting explicit unnest operators, CleanM avoids having to access repeating BLOB-like tuples of the form $(token_i, \{names\})$ for each element of a nested collection to be processed; it operates over smaller $(token_i, name_j)$ tuples instead [143]. Plan B checks the functional dependency: it computes groups of *address*, and outputs the groups containing more than one phone prefix. Plan C checks for duplicates by again building groups of *address* and checking within each group for entities that are more than 80% similar.

The algebraic rewriter of CleanM detects the commonalities of Plan B and C, and instead produces Plan BC, which coalesces the two grouping passes into one and applies both filters at once. In addition, given that all the sub-plans scan the same table, the algebraic rewriter produces a DAG-like overall plan, which scans the dataset once, performs the cleaning operations in parallel, and then joins the violating entries of each side using an outer join. In summary, translating cleaning operations into a unifying algebraic form enables, among others, powerful forms of query and data unnesting, coalescing operators, and reducing duplicate work.

6.5 Executing data cleaning tasks

The result of optimizations at the algebraic abstraction level of CleanM is a succinct logical plan. The last step of the rewriting process generates a physical plan that is compatible with the execution engine that will perform the data cleaning tasks. This work uses Spark [260] as the scale-out execution substrate; therefore, the algebraic plan gets translated to the operators of the Spark API.

Why not Spark SQL? Given that Spark is the current execution engine for CleanM queries, an alternative approach would be to directly map CleanM to the Spark SQL module of Spark [43], which exposes declarative query capabilities and introduces Catalyst, an optimizer over Spark. The Catalyst optimizer, however, assumes tabular data and only considers relational rewrites; it is thus unable to reason about and perform the optimizations suggested so far by this work.

Operator	Spark Equivalent
σ_p	filter
Δ_p^e	map → filter
μ_p^{path}	flatmap(x→path.filter(y → p(x, y)).map(y→(x,y)))
$\overline{\mu}_p^{path}$	flatmap(x→r=path.filter(y → p(x, y)), if(r.empty) (x, null) else r.map(y→(x,y)))
$\Gamma_p^{\oplus/e/f}$	**aggregateByKey** → mapPartitions
$\bowtie_{f(A)=g(B)}$	join
$\bowtie_{f(A)\ \theta\ g(B)}$	**theta join** → filter
$\overline{\bowtie}_{f(A)=g(B)}$	left outer join
$\overline{\bowtie}_{f(A)\ \theta\ g(B)}$	**theta join** → map

Table 6.1 – Translation of algebraic operators to Spark operators. Bold parts introduce new Spark operators or deviate from the translation that Spark SQL would have performed.

Also, the physical Spark plans that Catalyst generates are agnostic to characteristics of real-world data cleaning tasks, namely the facts that i) there is significant skew in the data touched, and that ii) the tasks executed typically require the computation of expensive theta joins. On the contrary, in the final, third abstraction level, CleanM queries get translated into a physical execution plan which both considers data skew and explicitly handles theta joins.

From nested algebra to Spark operators. Table 6.1 lists the mapping from the nested relational algebra to Spark operators. The mapping for the *selection* and *reduce* operators is straightforward. The *unnest* operators iterate through a dataset's elements and through a specific nested field of each element.

The *Nest* operator, which resembles a SQL *Group By*, is translated into a combination of operators: First, *aggregateByKey* groups data records based on a key. Then, *mapPartitions* applies a function over each partition. Nest optionally evaluates a binary predicate (an equivalent functionality to the SQL `HAVING` clause). In this case, a filter operation also takes place per partition. Finally, the Join operator gets translated into the respective Spark equi-join operator. The handling of other types of joins is more nuanced: By default, Spark SQL and Spark resort to a cartesian product followed by a filtering operation. Given the high frequency of theta joins in the domain of data cleaning, we instead implement an alternative, statistics-aware theta join [196].

Optimizations at the physical level

When translating nested relational algebra operators into a Spark plan, we explicitly consider the presence of i) skew in the data, and ii) theta joins as part of the cleaning process.

Handling data skew. Value distribution in real-world data is rarely uniform. In addition, certain data values can be more susceptible to errors. A cleaning solution must therefore be

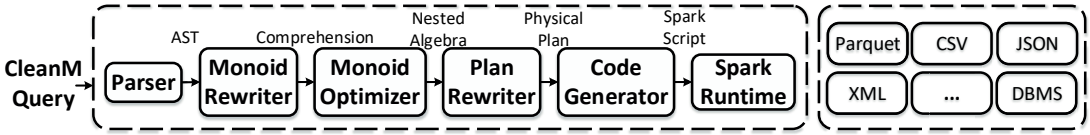


Figure 6.2 – The architecture of CleanDB.

resilient to data skew. In the context of scale-out processing, skew handling is reflected by how one shuffles data in the context of operations such as aggregations. Spark SQL performs sort-based aggregation: it sorts the dataset based on a grouping key, different data ranges of which end up in different data nodes. Then, Spark SQL performs any subsequent computations locally on each node. When, however, some values occur more frequently, the partitions created are imbalanced. Thus, the overloaded nodes lag behind and delay the overall execution. On the contrary, as Table 6.1 shows, CleanM uses the *aggregateByKey* Spark operator which performs the aggregate locally within each node and then merges the partial results. Thus, CleanM i) minimizes cross-node traffic by forwarding already grouped values, and ii) is more resilient to skew since popular values have already been partially grouped together.

Handling theta joins. In the general case of a join with an inequality predicate, Spark SQL generates a plan involving a cartesian product followed by a filter condition. The result is suboptimal performance when executing theta joins – one of the most frequent operators in data cleaning. We thus implement a custom theta join operator based on the approach of [196]. The new operator represents the cartesian product as a matrix, which it partitions into N uniform partitions. First, the operator computes statistics about the cardinality of the two inputs, which it then uses to populate value histograms. Then, assuming the presence of N nodes, the operator consults the observed value distributions to partition the matrix into N equi-sized rectangles, and assigns each partition to a Spark node. As a result, the operator ensures load balancing; each node checks separately the condition on the partition for which it is responsible.

6.6 CleanDB: A data cleaning system

We validate the three-level design of CleanM by implementing CleanDB, a unified cleaning and querying engine over Spark [260]. CleanDB serves as a replacement layer of Spark SQL [43]; it exposes the expressive power of CleanM without the compromises that Spark SQL makes. CleanDB optimizes the cleaning operations in a unified way and executes them in a scale-out fashion; the final physical plan is equivalent to handwritten Spark code. The end result is a system that can both query and clean input data.

The architecture of CleanDB. Figure 6.2 presents the components of CleanDB. When receiving a query, the *CleanM parser* rewrites it into an abstract syntax tree (AST). Then, the *Monoid Rewriter* "de-sugarizes" the AST into a monoid comprehension, also considering the monoids presented in Section 6.3. The *Monoid Optimizer* first applies rewrites over the input comprehension to simplify it, push down any filtering expressions, flatten nested comprehensions,

unnest existential quantifications, etc. Then, the optimizer rewrites the comprehension into a nested relational algebra, and performs additional rewrites and optimizations over it, such as coalescing multiple operators into a single one.

The output of the Optimizer is a nested relational algebra expression, which the *Physical Plan Rewriter* translates to a plan of physical operators. We plan to extend this level with more low-level "building blocks". Finally, the *Code Generator* dynamically generates the Spark script that represents the input query to reduce the interpretation overhead that hurts the performance of pipelined query engines [159]. After the generation of the Spark script, the Spark Executor deploys the final script in scale-out fashion.

Interestingly, Spark by default associates the result of the execution with the DAG of operations that produced it. We aim to use this built-in data lineage support to incorporate additional data cleaning functionality that considers data lineage [111] in future work.

6.7 Experimental Evaluation

The experiments examine how CleanDB performs compared to the state of the art, while demonstrating the benefits stemming from the three optimization levels of CleanM.

Experimental Setup. We compare CleanDB against BigDansing [149] because it is, to our knowledge, the only currently available scale-out system that explicitly targets data cleaning[1]. We also compare CleanDB against an implementation on top of Spark SQL. Spark SQL uses a relational optimizer to produce query plans, whereas CleanDB uses a monoid-aware, three-level optimizer; we can thus gauge the quality of the CleanM rewrites.

All experiments run on a cluster of 10 nodes equipped with 2 × Intel Xeon X5660 CPU (6 cores per socket @ 2.80GHz), 64KB of L1 cache and 256KB of L2 cache per core, 12MB of L3 cache shared, and 48GB of RAM. On top of the cluster runs Spark 1.6.0 – the latest version for which BigDansing is intended. Spark launches 10 workers, each using 4 cores and 40GB of memory.

The workload we use involves i) DC checks, ii) duplicate elimination, iii) term validation, and iv) syntactic transformations. DCs are a concept directly related to database design, thus we evaluate them over the TPC-H dataset. We use TPC-H for syntactic transformations as well. We use scale factors 15, 30, 45, 60, and 70 of the *lineitem* table. Each of the five versions comprises 90M, 180M, 270M, 360M, and 420M records and has size 11GB, 22GB, 34GB, 45GB, and 52GB respectively. We shuffle the order of the tuples and produce two different datasets by adding noise to 10% of the entries of the *orderkey* and *discount* column respectively. We pick the tuples to edit from the domain of the SF15 version, so that we increase the skew as we increase the dataset size. We also use a dataset which comprises tax information for people that live in the US [149]. We use the two versions of Tax used to evaluate BigDansing [149]; a 13-column version that contains FD violations, and a 4-column version that contains DC

[1] SampleClean [252] only operates over query-specific samples.

violations. Each version has a 100K and a 1M variation: The variations **100K-FD** and **100K-DC** have a size of 6.7MB and 2.3MB respectively, and the variations **1M-FD** and **1M-DC** have a size of 67MB and 26MB respectively.

We perform duplicate elimination and term validation over the DBLP bibliography hierarchical dataset, because these error categories occur frequently in semi-structured data. We use a subset of DBLP that contains information about articles; each entity contains at most 13 attributes. We add noise to 10% of the author names by a factor of 20%, and scale up the dataset by adding extra entities; we construct new publications by permuting the words of existing titles and by adding authors from the active domain. The end result is a 1GB, a 5GB, and a 10GB XML version. We also use the *customer* table of TPC-H because the implementation of duplicate elimination in BigDansing is a UDF that is specific to *customer*. We add duplicate records for 10% of customer entries, where the number of duplicates for each record is a random value generated using Zipf's distribution; the number of duplicates belongs to the intervals [1-50] and [1-100] respectively. We create the duplicate records by randomly editing the name and phone values. The size of the datasets is 2.2GB and 3.1GB respectively. We also use the Microsoft Academic Graph (MAG) [227], which is a database of scientific publications stemming from all research areas. We evaluate duplicate elimination over the original version of MAG, since its main issue is the existence of duplicate publications; the same publication may appear multiple times, with variations in the title and DOI fields, or with missing fields. We build MAG by joining the *Paper, Author* and *PaperAuthorAffiliation* datasets. The resulting dataset contains 7 columns and has size 33GB.

We use response time and accuracy (when applicable) as metrics. Response time includes the time taken to read the input, perform a cleaning task, and store the detected violations. In the case of term validation, the output includes both detected violations and suggested repairs. We measure accuracy by verifying the correctness of the repairs against a sanitized version of the dataset.

The rest of this section uses the aforementioned cleaning tasks to visit the CleanM optimization levels, and examines how each of them contributes to the fast and accurate responses of CleanDB.

6.7.1 Optimizations at the monoid level

CleanDB is the only scale-out cleaning system that supports term validation; Spark SQL would compute the cross product of the input and a dictionary, using a UDF to compute the similarity of each (record, dictionary value) pair, and prune non-similar entries. The overall Spark script was non-interactive in our experiments. This section demonstrates the benefits of monoid-level optimizations in the context of term validation; we examine clustering and filtering operations, and show the effect of calibrating each operation based on dataset characteristics.

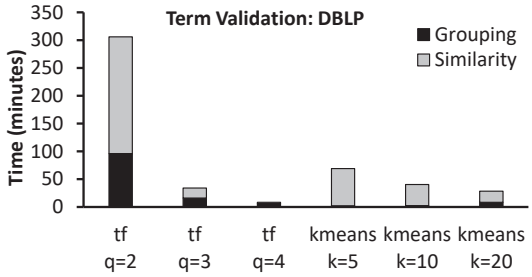


Figure 6.3 – Different configurations of CleanDB for term validation.

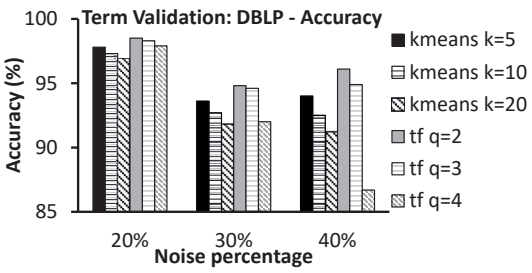


Figure 6.4 – Accuracy of term validation as the noise increases.

Term Validation

Term validation is a resource-intensive, challenging operation. This experiment validates the author names of the flat Parquet version of *DBLP* that contains 6.4M entities using the Levenshtein distance metric. The dictionary that CleanDB consults to repair author names comprises 200K names. The experiment launches different k-means configurations by changing the *number of centers (k)* which it obtains from the dictionary. The same experiment also launches different token filtering configurations using a different *token length parameter (q)*.

Runtime. Figure 6.3 presents the time taken to clean the author names using k-means and token filtering as pruning methods, while also using different parameters for each method. Each bar comprises the time taken to filter/block the data and the time to perform the similarity check within the groups. In the case of k-means, using more centers leads to fewer elements in each cluster. Thus, the number of similarity checks decreases. In the case of token filtering, as q increases, performance improves because the tokenization phase produces fewer groups with fewer elements in each one, and thus the number of checks decreases. The token filtering configurations are faster than the k-means ones, except when q=2; the token size proves to be too small and results in too many groups.

Regarding the pre-filtering step, since the tokenization process is expensive, grouping by center is more lightweight than grouping by token. However, the average length of author names in DBLP is 12.8, which is short enough for the tokenization to proceed without significant overhead. Regarding similarity checks, token filtering produces a larger number of smaller-sized groups compared to k-means, thus the total number of pairwise comparisons is smaller. K-means is more sensitive to the statically specified centers.

Accuracy. Table 6.2 measures the accuracy of the suggested repairs for the term validation task examined. The experiment considers `precision` (i.e., correct updates/total updates suggested), `recall` (i.e., correct updates/total errors) and `F-score` as metrics.

The token filtering configurations are more accurate, because they check the similarity of two author names whenever they have at least one common token. Thus, even if a name is dirty, it will contain at least one clean token that will match a token of the correct name

Type	Parameter(s)	Precision	Recall	F-score
tf	q = 2	100%	97%	98.5%
tf	q = 3	100%	96.8%	98.3%
tf	q = 4	99.9%	95.9%	97.9%
K-means	k = 5	99.9%	95.7%	97.8%
K-means	k = 10	99.9%	94.8%	97.3%
K-means	k = 20	99.9%	94%	96.9%

Table 6.2 – Accuracy of term validation approaches over the DBLP dataset.

in the dictionary. Increasing q does not hurt accuracy noticeably. K-means becomes less accurate as the number of clusters increases, because similar words end up in different clusters and therefore are not checked for similarity. Still, all the term validation variations of CleanDB exhibit high accuracy.

Figure 6.4 examines the accuracy of term validation as we vary the noise on the name attribute from 20% to 40%. To obtain a fair comparison, we lower the similarity threshold as we increase the noise, so that we isolate the accuracy of the pruning algorithm and avoid missing results that fail to pass the similarity threshold. The results show that accuracy drops slightly as we add more noise. The drop stems from both having lower precision and lower recall. Precision drops because some incorrect matches now pass the low similarity threshold; recall drops because by increasing the noise, two similar words are more likely to get assigned to different groups. However, the drop in accuracy is negligible in all cases but the ones where we have a bigger parameter set for token length q=4 or number of centers k=20; these configurations are more prone to inaccuracies because they produce clusters with fewer items.

Summary. CleanDB can use token filtering and clustering monoids to reduce term validation checks. Both methods avoid false positives, and thus the resulting precision is close to 100%. Calibrating the algorithm parameters enables trading performance for accuracy; still, the accuracy remains above 90% in most cases.

6.7.2 Optimizations at the algebra level

This section demonstrates the benefits of the algebraic optimizations that CleanDB performs. We focus on how CleanDB optimizes different cleaning operations as a single task.

Unified data cleaning

This experiment resembles our rolling example, and measures the cost of detecting duplicates and functional dependency violations through a single query on the customer dataset; we replace the term validation part of the example with an extra functional dependency, because CleanDB is the only scale-out system supporting term validation. The query in question examines the rules $FD1: address \rightarrow prefix(phone)$, $FD2: address \rightarrow nationkey$ and also

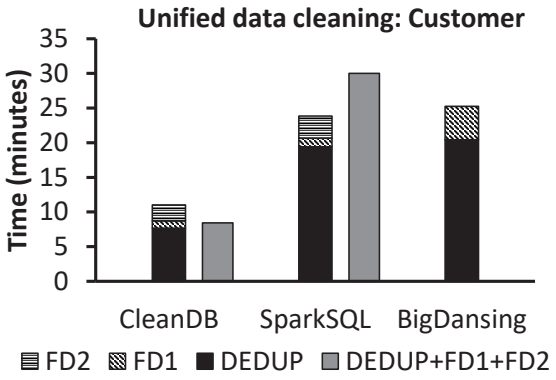


Figure 6.5 – Unified data cleaning: CleanDB rewrites three cleaning operations into a single one, and avoids duplicate work.

checks for duplicate customers given that they appear with the same address. We run the query as i) separate sub-queries and ii) as a single task that also combines the partial results. Figure 6.5 presents the results.

Results. CleanDB detects that the tasks share a grouping on the *address* field and performs all operations using a single aggregation step. Unifying the cleaning tasks reduces the execution time for CleanDB. BigDansing can only apply one operation at a time and lacks support for values not belonging to the original attributes (i.e., the result of *prefix()* in *FD1*). Spark SQL is unable to detect the opportunity to group the tasks into one. It starts the cleaning tasks in parallel since they share a common data scan, but then performs a full outer join to combine the output of each operation; unified execution ends up being more expensive than the standalone one. Still, even considering the separate execution, CleanDB outperforms the other systems because of its explicit skew handling when performing FD checks and deduplication.

Transformations

This experiment measures the cost of applying syntactic transformations over the SF70 Parquet version of TPC-H. The experiment examines the added cost when performing lightweight cleaning tasks compared to a traversal of the dataset that projects all its attributes. We consider filling missing values and splitting dates. We fill empty values of the *quantity* attribute using the average value of the existing quantities. We split the *receipt_date* into *day*, *month* and *year* fields. We also measure the cost of applying the aforementioned operations using a single CleanM query.

Results. Table 6.3 shows the slowdown that each cleaning task incurs compared to executing the plain query. The individual costs of splitting the dates and filling missing values are almost masked by the query cost. When applying each cleaning operation one after the other, the overall slowdown is computed by adding the overall running times for each dataset traversal. However, CleanDB is able to apply both cleaning operations in one go: The overall cost is then similar to the cost of only applying a single operation, because the execution plan computes the average quantity and then performs both the replacement of missing values and the

Operation	Slowdown
Split date	1.15×
Fill values	1.15×
Split date & Fill values (two steps)	2.3×
Split date & Fill values (one step)	1.19×

Table 6.3 – Overhead introduced by performing syntactic transformations in a plain query. The optimizer of CleanDB applies both operations in one go and reduces overhead by ~ 2×.

splitting of the receipt column in a single dataset pass. In summary, CleanDB can intertwine analytics and lightweight cleaning operations, while relying on its optimizer to identify and prune duplicate work.

Summary. Instead of treating each type of cleaning operation as a standalone implementation, CleanDB optimizes a cleaning workflow as a whole, identifying optimization opportunities even across different operations. CleanM enables such optimizations because it uses a single abstraction to express all cleaning tasks, and an optimizable algebra as its backend.

6.7.3 Optimizations at the physical level

This section shows how the physical-level optimizations of CleanDB that focus on handling skew and non-equality predicates accelerate data cleaning and duplicate elimination tasks.

Functional Dependencies & Denial Constraints

This experiment measures the cost of validating four rules; ϕ_1 and ψ_1 concern TPC-H, while ϕ_2 and ψ_2 concern Tax. Rule ϕ_1 is a functional dependency (FD) stating that the order of an item determines its supplier. Rule ψ_1 is a denial constraint (DC) stating that an item cannot have a bigger discount than a more expensive item; the filter on *price* has a selectivity of 0.01%. Rule ϕ_2 is a FD stating that the zip code determines the city and state, and ψ_2 is a DC stating that the tax must be analogous to the salary of an employee.

$$\phi_1 : orderkey, linenumber \rightarrow suppkey, \phi_2 : zip \rightarrow city, state$$
$$\psi_1 : \forall t_1, t_2\ t_1.price < t_2.price\ \&\ t_1.discount > t_2.discount \\ \&\ t_1.price < [X]$$
$$\psi_2 : \forall t_1, t_2\ t_1.salary < t_2.salary\ \&\ t_1.tax > t_2.tax$$

The straightforward way to detect FD violations using (Spark) SQL is a self-join query. However, traversing a dataset twice hurts performance. Thus, we benchmark FDs in Spark SQL using a query that groups the data in a way similar to CleanM. To collect the distinct values per group, we implement a user-defined aggregate function that behaves similar to `GROUP_CONCAT`.

TPC-H FD Results. Figures 6.6a, 6.6b present the time taken to detect violations of ϕ_1 as we increase the size of TPC-H. We present the results for both CSV (Figure 6.6a) and Parquet

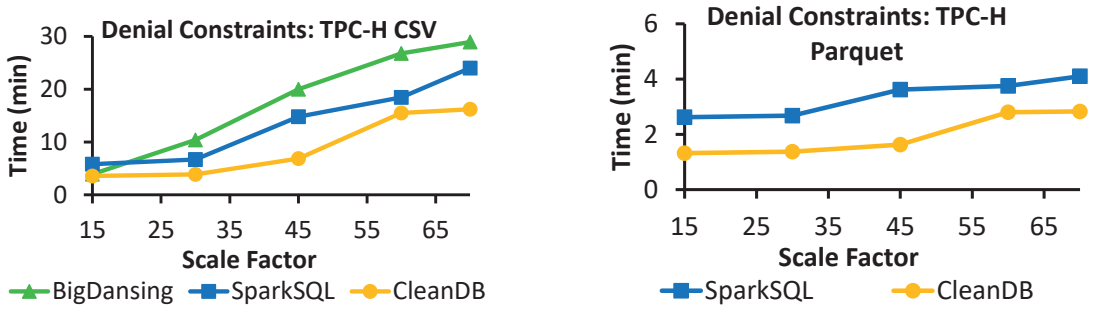


Figure 6.6 – Cost of checking for violations of functional dependencies over TPC-H.

(Figure 6.6b). Parquet is only supported by CleanDB and Spark SQL; we omit BigDansing in Figure 6.6b. The response times of Figure 6.6b are shorter than those of Figure 6.6a because Parquet is a binary columnar optimized data format which also supports compression.

CleanDB is faster than BigDansing and Spark SQL regardless of the underlying format. BigDansing performs hash-based aggregation: it shuffles the data based on a hash function to create blocks that share the same *orderkey* and *linenumber*, and then iterates through each block to check for violations. Spark SQL performs sort-based aggregation: it sorts the entire dataset based on the (*orderkey*, *linenumber*) pair, and different data ranges end up in different data nodes. Then, it performs the aggregate computations locally on each node. Spark SQL outperforms BigDansing because the sort-based shuffle implementation of Spark is more efficient than the hash-based one [256]: The hash-based approach stresses the overall system memory and causes a lot of random I/O, whereas the sort-based approach uses external sorting to alleviate these issues. CleanDB considers data skew when creating the physical query plan: It performs the aggregate operation locally within each data node and then merges the partial results, thus minimizing cross-node traffic. Therefore, CleanDB outperforms the other systems because it translates the query into a set of Spark operators that do not require data exchange until the final merge phase.

Scale Factor	15	30	45	60	70
Time (min)	1.7	2	3.7	4.9	5.65

Table 6.4 – Denial constraints involving inequalities as the dataset size increases. All systems beside CleanDB fail to terminate.

TPC-H DC Results. The detection of violations of ψ_1 involves a self-join that checks the inequality conditions. Table 6.4 shows that only CleanDB was able to successfully complete the data constraint check. Spark SQL was unable to compute the expensive cross product to evaluate the conditions. BigDansing and CleanDB rely on a custom theta join operator each. The theta join implementation of BigDansing attempts to prune the pairwise comparisons involved in the computation of an inequality join by first partitioning the data, then computing min-max values per partition, and then only cross-comparing partitions whose min-max

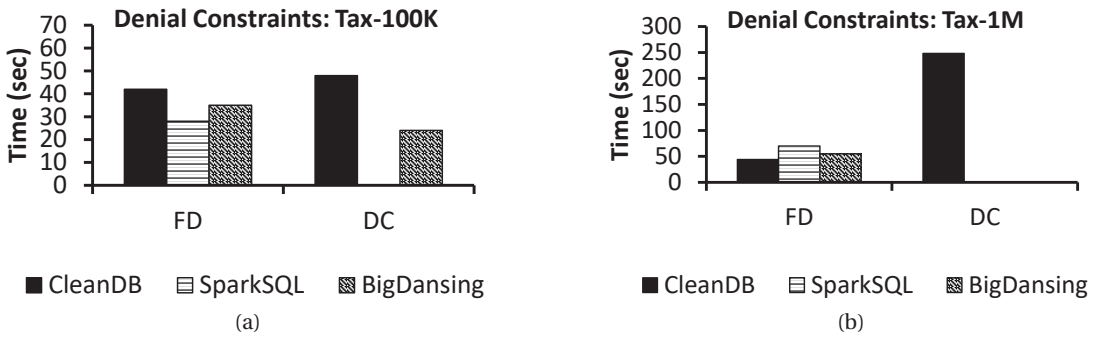


Figure 6.7 – Cost of checking for violations of functional dependencies over Tax.

ranges overlap. The number of avoidable checks, however, is not guaranteed to be high, unless the partitioning of the first step can be fully aligned with the fields involved in the DC; indeed, excessive data shuffling makes BigDansing non-responsive for ψ_1. On the contrary, CleanDB spends more effort to obtain global data statistics and does a better job balancing the theta join load among the Spark executors.

Tax Results. Figures 6.7a, 6.7b show the time taken to detect violations of rules ϕ_2, ψ_2 over the Tax dataset. When evaluating the FD ϕ_2 over the **100K-FD** version of Tax, the input size is too small for the skew-balancing optimizations of CleanDB to prove useful. **100K-DC** is even smaller (only 2.3MB). In addition, the DC version of Tax of [149] is a synthetic variation that contains a small number of violations. Therefore, the effort of CleanDB to create balanced data partitions for scale-out execution does not pay off. Still, even for this small size, SparkSQL is unable to terminate because of the cartesian product it attempts to evaluate.

As shown in Figure 6.7b, CleanDB scales better than its competitors: CleanDB is the fastest system over **1M-FD**, and the only system that successfully terminates for **1M-DC**, because it balances the load more uniformly by aggregating the results locally.

Duplicate Elimination

The following experiments evaluate duplicate detection using DBLP, MAG, and the TPC-H customer table; the duplicate elimination implementation of BigDansing is specific to the customer table.

We demonstrate the importance of being able to handle heterogeneous datasets by considering different representations for DBLP: We consider i) a JSON version, which has become the most popular data exchange format, ii) a Parquet version that preserves data nestings, iii) a "flat" CSV version, and iv) a "flat" Parquet version. We obtain the last two versions by flattening the entities of the nested input; if a publication has more than one author, then the publication appears in multiple records – one for each author. We compare the response time of CleanDB against Spark SQL. We consider two DBLP publications to be duplicates if they appear on the same journal, have the same title, and the similarity of their attributes

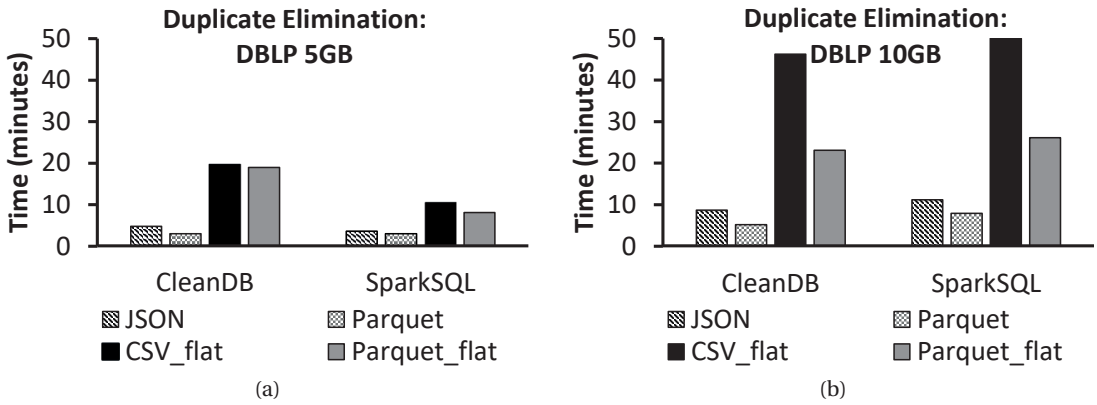


Figure 6.8 – Duplicate elimination over simplified representations of DBLP: Spark SQL was unable to terminate when cleaning the original dataset.

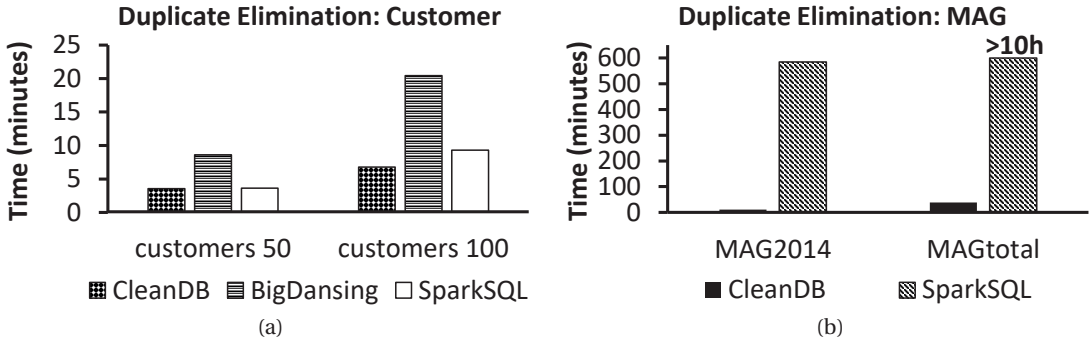


Figure 6.9 – Duplicate elimination over Customer and MAG.

exceeds 80% – we assume that the title and journal attributes are "cleaner" than the rest. Both CleanDB and Spark SQL create blocks based on the journal and title values to reduce pairwise comparisons. Similarly, two MAG publications are duplicates if they appear on the same year, have the same author id, and are more than 80% similar.

DBLP Deduplication Results. Spark SQL initially was unable to complete the elimination task, even for an input size of 1GB, because it is sensitive to data skew. Therefore, we removed the most frequently occurring titles from the dataset to obtain a more uniform version and enable the comparison against Spark SQL. The size of the uniform dataset varies from 5GB to 10GB when stored as XML, and the number of entries ranges from 6.4 to 64 million. For the JSON, nested Parquet, "flat" CSV, and "flat" Parquet versions, the size reached 7GB, 2GB, 14GB, and 2.4GB respectively.

Figure 6.8 presents the response time of the systems that are able to process DBLP. Both CleanDB and Spark SQL are faster when running over the nested JSON and Parquet representations, because flattening the data introduced many more tuples to be processed; thus, being able to operate over the original, non-relational data representation can be a significant asset for many use cases.

Regardless of format, Spark SQL exhibits lower response times for the 5GB case, yet scales less gracefully and is slower than CleanDB for the 10GB version. The explanation for this behavior resembles the one for DCs: Spark SQL uses sort-based shuffling based on the *journal, title* attributes to assign the records of each group into the same partition and then computes the similarity within each group. On the contrary, CleanDB aggregates data locally, and then merges the partial results together. The physical rewrites of CleanDB reduce network traffic and are resilient to skew. However, in the simplified dataset versions that we produced to be able to use Spark SQL, data ends up following a uniform distribution, thus favoring Spark SQL. Still, when the data size increases, some of the values again occur more frequently than others; Spark SQL creates imbalanced partitions, which overload some nodes and thus delay the overall execution time because they have to perform more similarity checks than other nodes.

Customer Deduplication Results. Figure 6.9a presents the response time of all systems over the customer dataset. BigDansing and Spark SQL perform poorly because of the suboptimal way in which they construct the value blocks to be checked for duplicates; instead of grouping values locally and then shuffling them to other nodes, they shuffle the entire dataset. CleanDB scales better than the other systems because of its explicit skew handling.

MAG Deduplication Results. Figure 6.9b presents the response time of all systems over the MAG dataset. Spark SQL was unable to execute the task for the whole dataset, thus we also consider a 6.3GB subset that contains publications from year 2014. MAG is a real-world, highly skewed dataset; CleanDB uses skew-resilient primitives, and thus significantly outperforms Spark SQL.

Summary. The physical-level optimizations, namely support for data skew and theta joins, ensure that CleanDB scales gracefully, and handles realistic datasets for which its competitors are unable to terminate successfully. The experiments also show the importance of allowing data cleaning over the original, intended data format; cleaning nested data proved to be faster when considering the original nested representation instead of flattening all entries.

6.8 Summary

Practitioners typically perform manual data cleaning or use a number of cleaning tools – one per error type. Being forced to use multiple tools is inconvenient, makes it hard to apply cleaning operations iteratively until the user considers data quality to be satisfactory, and seldom guarantees that a cleaning script will be efficiently optimized and executed as a whole.

This work introduces CleanM, a declarative query language that allows users to express their different cleaning scripts. CleanM exposes a wide variety of parameterizable data cleaning primitives, which a user can apply over her data. CleanM relies on a powerful, parallelizable query calculus, and a three-level optimization process; all the operations included in a cleaning script are translated to the calculus, and then optimized as one unified task.

We have implemented CleanDB, a scale-out querying and cleaning framework. CleanDB exposes the functionality of CleanM over multiple types of data sources. CleanDB scales better than existing data cleaning solutions and handles cases that other systems lack support for or are unable to serve due to performance issues.

7 Looking forward: HTAP on Heterogeneous Hardware

Modern database engines balance the demanding requirements of mixed, hybrid transactional and analytical processing (HTAP) workloads by relying on i) global shared memory, ii) system-wide cache coherence, and iii) massive parallelism. Thus, database engines are typically deployed on multi-socket multi-cores, which have been the only platform to support all three aspects.

Two recent trends, however, indicate that these hardware assumptions will be invalidated in the near future. First, hardware vendors have started exploring alternate non-cache-coherent shared-memory multi-core designs due to escalating complexity in maintaining coherence across hundreds of cores. Second, as GPGPUs overcome programmability, performance, and interfacing limitations, they are adopted by emerging servers to expose heterogeneous parallelism. It is thus necessary to revisit database engine design because current engines neither deal with the lack of cache coherence nor exploit heterogeneous parallelism.

In this chapter, we make the case for Heterogeneous-HTAP (H^2TAP), a new architecture explicitly targeted at emerging hardware. H^2TAP engines store data in shared memory to maximize data freshness, pair workloads with ideal processor types to exploit heterogeneity, and use message passing with explicit processor cache management to circumvent the lack of cache coherence.

7.1 Introduction

The past few years have witnessed a rise in demand for real-time business intelligence. Organizations increasingly require analytics on fresh operational data to derive timely insights. To meet these requirements, database engines have to efficiently support hybrid transactional and analytical workloads (HTAP) over shared data. Designing a database engine that can serve mixed workloads efficiently is challenging, because OLTP workloads require ACID semantics, high throughput, and performance isolation, while OLAP workloads require interactive response times and data freshness.

Database engines meet these conflicting demands by relying on hardware to support three important functionalities. First, they rely on global shared memory to store a single copy of data that can be accessed by both OLTP and OLAP workloads. Second, they rely on cache coherence to guarantee that two threads running on different cores see a consistent view of data stored in shared memory despite layers of caching. Third, they rely on abundant parallelism to concurrently execute OLTP and OLAP queries. Despite providing massive parallelism, accelerators like GPGPUs have traditionally neither shared memory nor maintained coherence with CPUs. Thus, contemporary database engines are designed to be deployed on high-end multi-socket multi-cores.

Two recent trends, however, necessitate revisiting contemporary database engine design. First, as we move from the multi-core era to the many-core one, maintaining coherence across hundreds of core-private caches has become challenging. Architecture researchers and hardware vendors have started exploring many-core designs that support global shared memory but not system-wide cache coherence [49, 50, 131, 178, 253]. Second, over the past few years, GPGPUs have evolved from memory-limited, niche accelerators into general-purpose processors that support, among other advanced features, globally shared address space and pageable virtual memory. Based on these trends, emerging hardware will likely have three salient properties: i) heterogeneous parallelism, ii) global shared memory, and iii) no system-wide cache coherence. Current database engines are a poor match for emerging hardware because they can neither deal with the lack of cache coherence nor exploit heterogeneous parallelism. As a result, despite underutilizing hardware resources, current engines deployed on emerging hardware will continue to suffer from a "house pattern" [210]: OLTP and OLAP workloads will negatively interfere with each other due to resource contention.

This chapter presents Heterogeneous-HTAP (*H^{2}TAP*), a new architecture for designing database engines explicitly targeted at emerging hardware. The H^{2}TAP architecture requires database engines to address all three aspects of emerging hardware explicitly by adhering to two design principles: i) make heterogeneity a first-class design citizen, ii) decouple shared memory from cache coherence. Using these principles, H^{2}TAP database engines exploit heterogeneity by pairing processors with their ideal workloads, provide data freshness for OLAP workloads by storing data in globally shared memory, and use message-passing-based parallelism instead of shared-memory parallelism to scale OLTP workloads even in the absence of cache coherence. We validate the H^{2}TAP architecture by designing and implementing Caldera, a prototype H^{2}TAP engine. Our evaluation shows that Caldera can provide transactional throughput comparable to state-of-the-art OLTP engines while providing interactive response time and data freshness for analytical queries using GPGPUs.

7.2 Database engines on emerging hardware

As also described in Section 2.4, the hardware landscape exhibits two major trends to which the data management sector must adapt, namely, the generalization of GPGPUs and the

specialization of multisocket CPUs. GPGPUs have evolved from memory-limited accelerators for niche computations to general-purpose processors, whereas architecture researchers and practitioners have started exploring specialized multicore CPU designs.

Putting the hardware trends together, we believe that in the near future, the servers that will be used to deploy database engines will have three salient properties: 1) they will support heterogeneous parallelism with CPUs that excel at latency-critical *task-parallel* workloads and GPGPUs that excel at throughput-heavy *data-parallel* workloads, 2) similar to contemporary servers, they will support a global address space that is shared across all processors, and 3) unlike contemporary servers, they will not support system-wide CC. Current engines suffer from three major problems on such hardware.

First, database designs that rely on CC-shared memory for scaling transactional workloads will be incompatible with non-CC hardware. Database engines rely on CC for cross-core data sharing, and more importantly, thread synchronization based on spinlocks, shared-memory atomics, or HTM. In the absence of system-wide CC, the only option today is to scale OLTP workloads using the shared-nothing (SN) design. The SN design, however, is agnostic to the fact that memory is globally shared across all processors, and thus suffers from distributed transaction overheads when running poorly partitionable workloads [207].

Second, while specialized OLAP engines exploit the massive parallelism of GPGPUs [15, 127, 128], all current general-purpose engines ignore them because they traditionally did not share an address space with CPUs, and thus made it difficult to share data across transactional and analytical workloads. Using these contemporary database engines on emerging hardware with GPGPUs that no longer suffer from any such data-sharing limitations would leave abundant heterogeneous parallelism untapped.

Third, even state-of-the-art database engines exhibit a *house pattern* [210]: under mixed workloads, increasing OLAP throughput by scheduling more concurrent analytical queries results in a collapse in transactional throughput due to contention for processing resources. Avoiding the house pattern requires throttling or preempting analytical queries in order to prioritize transaction execution. Such throttling is completely unwarranted in emerging server platforms, especially since the heterogeneous processing resources are underutilized. Given these problems, we believe that it is time to revisit database design for emerging hardware.

7.3 The case for H^{2}TAP

Heterogeneous-HTAP (H^{2}TAP) is a new architecture for building database engines that uses two design principles to exploit all aspects of emerging hardware: 1) make heterogeneity a first class design citizen, 2) decouple shared memory and CC dependencies.

Heterogeneity as an opportunity. The H^{2}TAP architecture exploits heterogeneity based on the observation that the latency-critical nature of OLTP workloads and the bandwidth-

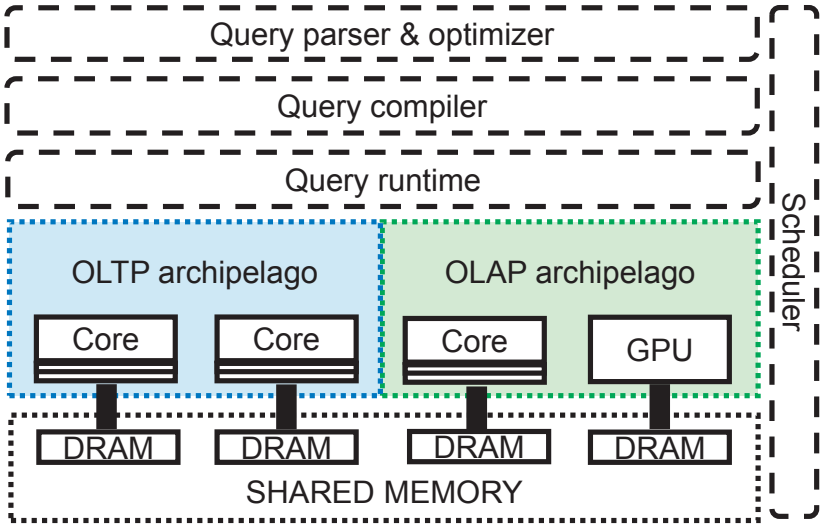


Figure 7.1 – H^{2}TAP deployed over emerging server hardware.

intensive nature of OLAP workloads are aligned with the task-parallel nature of CPUs and the data-parallel nature of GPUs respectively. Thus, the H^{2}TAP architecture uses both hardware and workload heterogeneity in a synergistic fashion by introducing the *archipelago* abstraction.

Archipelagos are resource containers defined by a set of processor cores and a target workload. H^{2}TAP uses archipelagos by partitioning cores into a task-parallel archipelago consisting only of CPU cores, and a data-parallel archipelago that can contain both GPUs and CPU cores. Transactions are executed in the task-parallel archipelago while analytical queries are handled by the data-parallel archipelago as shown in Figure 7.1.

Decoupling shared memory and cache coherence. Despite executing queries in different archipelagos, the H^{2}TAP architecture mandates storing a single copy of data in shared memory that is *globally accessible* across archipelagos. Still, while the H^{2}TAP architecture expects hardware to support shared memory, it does not rely on system-wide CC. Instead, H^{2}TAP engines have to explicitly manage CC in software. The clear separation of workloads across archipelagos simplifies this task to a certain extent – as analytical queries do not update data, H^{2}TAP engines do not have to maintain coherence across archipelagos. However, H^{2}TAP engines should guarantee that transactions running within the task-parallel archipelago obey the ACID properties and analytical queries running in the data-parallel archipelago work on transactionally consistent data, despite the lack of CC.

H^{2}TAP blueprint. Figure 7.1 shows software components that an H^{2}TAP engine would need to implement in order to realize the H^{2}TAP architecture in practice. The *parser and optimizer* form a front-end that translates a SQL query into a physical query plan. The *scheduler* is responsible for implementing the archipelago abstraction by managing core–archipelago membership. Using this information, the scheduler can provide run-time elasticity by enabling on-the-fly "migration" of CPU cores between archipelagos. Further, the scheduler also

maintains processor and memory utilization statistics within each archipelago. Based on these statistics, it works with the optimizer to determine the target archipelago and cores where each query will be executed. While the H^{2}TAP architecture requires transactional queries to be scheduled on CPUs in the task-parallel archipelago, it enforces no such restrictions on the scheduling of analytical queries in the data-parallel archipelago. Thus, the scheduler can combine dynamic run-time information, such as data locality, with static optimizer cost models to decide if a given analytical query should be executed on CPU or GPU cores in the data-parallel archipelago.

Once the scheduler determines the target execution environment, a *query compiler* produces the query implementation from the physical query plan. Instead of using volcano-style interpretation for executing the query plan, the query compiler generates machine code corresponding to the target processor(s) for the query. Query compilation reduces the interpretation overheads of query execution [153, 159, 190, 213, 153] and masks the effects of (data) heterogeneity [143, 144, 145]; H^{2}TAP extends the concept of heterogeneity to hardware to mask the difference in Instruction Set Architectures (x86 or PTX [194]).

Finally, the generated code is passed to the *Query runtime* together with information from the scheduler about the target processor(s) where the query should be executed. The runtime is responsible for both providing a mechanism for sharing data across archipelagos and shepherding query execution within each archipelago.

H^{2}TAP benefits. The H^{2}TAP architecture provides several benefits. First, archipelagos enable affinitizing workloads to ideal processor types; transactions benefit from task parallelism provided by CPUs and analytical queries benefit from data parallelism provided by GPUs. Second, by enabling CPU cores to change membership between task and data-parallel archipelagos on the fly, the H^{2}TAP architecture improves deployment elasticity because it enables dynamic load balancing. For instance, an H^{2}TAP engine could configure its scheduler to move unused CPU cores from task- to data-parallel archipelago, and use them for running analytical queries under light OLTP workloads. Third, by separating OLTP and OLAP execution, archipelagos eliminate interference and processor resource contention across workloads, and hence the house pattern, by design. Fourth, by decoupling shared memory and CC, the H^{2}TAP architecture enables new database engine designs that can take a middle ground between shared-everything designs, which rely on CC and shared memory, and shared-nothing designs, which are oblivious to both aspects.

H^{2}TAP challenges. Despite the benefits of H^{2}TAP, realizing it in practice also requires answering three questions. First, H^{2}TAP engines have to store data in a layout that is suitable for efficiently running both transactional and analytical workloads. However, research on CPU-based database engines has shown that different workloads benefit from different storage layouts [33, 86]. OLTP workloads benefit from the N-ary Storage Model (NSM) because the whole-record read-write operations performed by transactions can be implemented efficiently using NSM's row-wise layout. OLAP workloads, in contrast, touch only a few attributes, and

thus benefit more from the columnar layout of the Decomposition Storage Model (DSM), which minimizes data transfers and utilizes CPU caches better. As H^{2}TAP engines need to support both workloads, the first question to be answered is whether "middle-ground" [33] hybrid layouts [37, 44, 120] work in the H^{2}TAP context as well.

Second, irrespective of the layout used, an H^{2}TAP engine must provide an efficient mechanism to provide analytical queries running in the data-parallel archipelago with access to transactionally-consistent data, which is being updated by transactions running on CPUs. Contemporary HTAP engines typically use snapshotting to solve this problem [147]. If we used only CPUs in the data-parallel archipelago, we would be able to use fork-based snapshotting [147] for executing OLAP queries over an immutable database snapshot. Unfortunately, such an approach is not applicable with GPGPUs because CUDA memory allocations cannot be shared across process boundaries due to CUDA runtime limitations. Thus, the second question to be answered is whether alternate software snapshotting techniques [248] can be used to enable cross-archipelago data sharing.

Third, while the H^{2}TAP architecture expects hardware to support globally accessible shared memory, it does not rely on system-wide CC. Thus, an H^{2}TAP engine must be able to scale transactional and analytical workloads despite the lack of CC. Given that OLAP queries running in the data-parallel archipelago never update the database due to their read-only nature, H^{2}TAP obviates the need for cross-archipelago CC. However, H^{2}TAP engines must still overcome the lack of coherence within the task-parallel archipelago where concurrent transactions update shared data and metadata. Therefore, the third question to be answered is whether OLTP workloads can be scaled up within task-parallel archipelagos without relying on CC.

7.4 CALDERA: An H^{2}TAP query engine

Caldera is a prototype query engine we develop to examine the opportunities offered by the H^{2}TAP architecture and address the challenges it raises. To this end, the Caldera prototype implements only the query runtime and leaves the other components described in Section 7.3 to future work.

Applying H^{2}TAP. Caldera adheres to the H^{2}TAP architecture by grouping processors into a CPU-only task-parallel archipelago, and a GPU-only data-parallel archipelago. Transactions are executed in the task-parallel archipelago while analytical queries are handled by the data-parallel archipelago.

Caldera stores data in shared memory that is allocated using Unified Virtual Addressing. By using UVA, Caldera exposes a global address space across archipelagos. We use UVA because our current hardware setup uses Maxwell GPUs, which impose strict limits on the maximum Unified Memory allocation size. In the future, we plan to use Unified Memory with Pascal GPUs that have no such limitations.

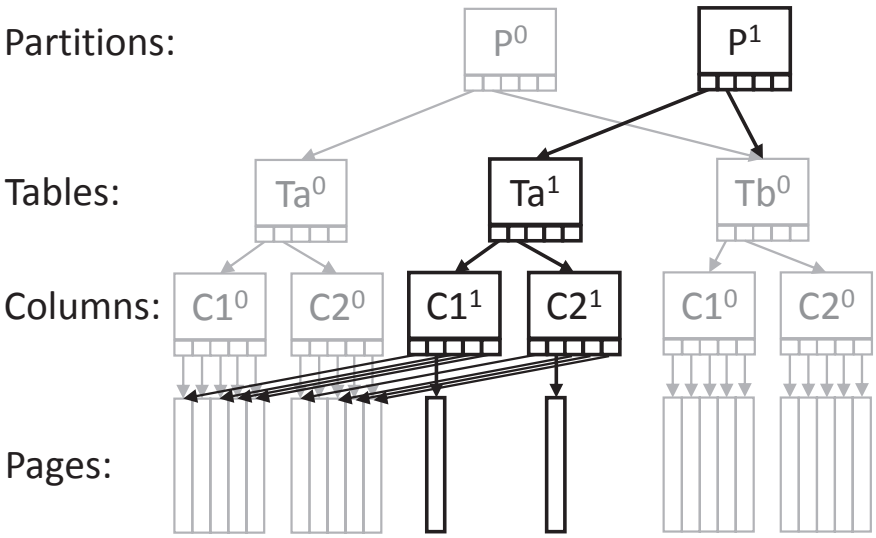


Figure 7.2 – The hierarchical data organization of Caldera for a columnar data layout, and the in-memory state after a transaction has updated table T^a. Superscripts represent epochs.

Data layout. Prior research has focused on building hybrid layouts that can support both transactional and analytical workloads in the traditional HTAP context [33, 37, 44, 120]. For instance, PAX [33] is an alternative storage layout that strikes a balance between the NSM and DSM extremes. Like NSM, PAX organizes data records in pages. Like DSM, PAX groups values of the same attribute together. A page therefore contains *minipages*, each of which only contains values of a single attribute. Due to its organization of data into minipages and pages, PAX enables cache-friendly query execution similar to DSM while providing update cost similar to NSM.

Hybrid layouts like PAX play an even more important role in the new H^{2}TAP scenario because they provide two tangible performance benefits. First, as the GPU memory capacity is limited, data transfer plays a crucial role in determining the overall query execution time due to the limited bandwidth of the PCIe bus. Thus, hybrid layouts will outperform NSM even in the H^{2}TAP scenario due to their ability to reduce the amount of data transferred. Second, GPUs coalesce global memory loads and stores issued by threads into as few memory transactions as possible to both improve performance and reduce memory bandwidth requirements. However, in order for coalescing to work properly, threads should access memory locations sequentially. Thus, a data layout like PAX is a better fit for GPUs than NSM because it enables such coalesced accesses.

Our current prototype supports NSM, DSM, and PAX layouts. Caldera stores data in shared memory as a collection of horizontal *partitions*. Within a partition, records of a table can be stored in any of the three layout types. Figure 7.2 shows the hierarchical *partition–table–column–page* data organization used by DSM.

OLAP in the data-parallel archipelago. The Caldera prototype uses the kernel-based execution model for executing OLAP queries on the GPU similar to other GPU-based OLAP engines [96, 127, 258]. Each database operator is implemented as a collection of data-parallel primitives, where each primitive is an individual CUDA kernel. OLAP queries are executed by a dedicated CPU thread that executes each database operator by executing the corresponding CUDA kernels one at a time while using UVA to store all input, intermediate, and output data. It is well-known that such kernel-based execution results in sub-optimal use of the GPU due to unwarranted data transfers [205, 258]. In the future, we plan to use a query compilation infrastructure to fuse multiple relational operators in a single kernel.

Caldera always executes OLAP queries on a database snapshot. Thus, users can trade off data freshness for performance by having several OLAP queries share a snapshot, or maximize freshness by taking a snapshot before running each OLAP query. Snapshotting is implemented using a software-based shadow-copying mechanism that works on the hierarchical data organization. We describe it using the layout shown in Figure 7.2. Each table, column, and page is associated with an epoch number. The query runtime creates a snapshot by performing a shallow copy of the top-level container and incrementing its epoch number. Thus, snapshotting is an instantaneous operation after which the newly created snapshot and the "live" database share all data. After snapshotting, the runtime identifies the columns that are necessary for executing the OLAP query and invokes the GPU kernel, passing in pointers to relevant pages. No data is copied explicitly; the GPU kernel accesses data directly from the UVA-allocated host memory.

Copy-on-write during updates and garbage collection are integrated with transaction management. When a transaction commits, the runtime identifies records to be updated. It uses this information to identify the backing pages for those records, and shadow-copies them by allocating new pages and copying over data from the snapshot. Then, it applies the updates, and marks the pages as "live" by incrementing their epoch number. It repeats this copy-on-write process all the way back to the root, allocating new data structures as required and updating pointers. Similarly, when a snapshot is deleted, the query runtime uses epoch numbers to identify both data and metadata that have been superseded by the copy-on-write process and deletes the old versions to reclaim space.

OLTP in the task-parallel archipelago. Caldera scales OLTP workloads within the task-parallel archipelago by using message passing-based parallelism (that relies on fast core-to-core messaging) rather than shared-memory parallelism (that relies on cache coherence). Caldera schedules one thread per core in the task-parallel archipelago and assigns one data partition to each thread, which then mediates access to partition-local records. Each thread uses two-phase locking (2PL) for concurrency control and a primary-key index to assist in record lookup. Unlike data, which is shared across archipelagos, the lock tables and indices are private to each thread running in the task-parallel archipelago and do not belong to the snapshot hierarchy depicted in Figure 7.2. Thus, they refer to logical records whose physical location changes during copy-on-write operations.

An incoming transaction can be scheduled to run on any thread; the chosen thread will act as its host (the *client* thread). The client executes all operations of a transaction using direct function calls to lookup/update records. If the client contains the target record in its partition, it uses its local lock table to decide if the access request can be granted. If so, it grants the lock, performs shadow copying if necessary, and executes the operation.

If the record belongs to a different partition, the client sends a message to the data owner thread (the *server* thread) requesting access to the record, and blocks the transaction. When the server thread receives the message, it tries to acquire the lock. If successful, it grants the lock, performs shadow copying if necessary, and sends a reply message giving the client access to the record. If the acquisition fails, the server thread delays replying back until the lock becomes available. Rather than shipping the whole record in the message, Caldera exploits hardware-supported shared memory to reduce data movement by sending only the record pointer. Upon receiving the reply, the client thread unblocks the transaction and uses the record pointer to directly lookup/update the record. At transaction commit or abort time, the client thread sends an explicit "release" message for each remote record. Upon receiving a release message, the server thread releases the associated lock and picks a new lock owner. If the new owning transaction is local to the server, it is unblocked and scheduled for execution. Otherwise, the server unblocks it by replying back to the client.

Relying on explicit message passing has several benefits. First, two processors can never simultaneously access a shared memory word because each processor has exclusive access over its partition. Before a thread can access a record, it has to explicitly synchronize with the owning thread by sending it a message. This explicit communication eliminates the need for implicit thread synchronization with latches, atomics, or other CC-dependent hardware features. Thus, all aspects of transaction execution are single-threaded and completely synchronization-free.

Explicit communication also makes maintaining coherence across core-private caches straightforward. In Caldera, two transactions can never concurrently update the same record due to 2PL. Thus, cache management is necessary only to ensure that two transactions running serially on two different cores see the latest version of the record despite the existence of caches. This can be done by adding explicit cache write back and invalidation at two points. When a client thread requests a record from a server thread, the server thread explicitly writes back the dirty data from its local cache before replying back. Similarly, before the client thread sends a release message at commit time, it writes back the data it updated. Doing so guarantees that a thread will always read the latest version of data from the memory instead of an outdated cache. Together, explicit communication and cache management ensure that Caldera can work on non-CC hardware.

Finally, by abstracting away the details of communication using a message passing library, Caldera is portable, as the message-passing layer can be replaced to make it work on CC multicores, non-CC multicores, and even potentially scale-out clusters without any change to the core database logic.

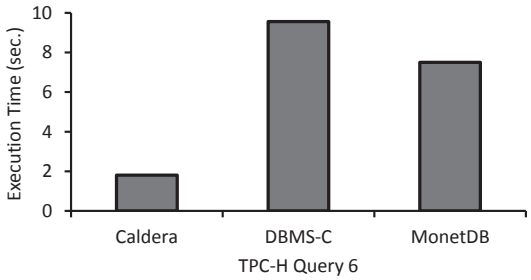


Figure 7.3 – GPU-powered Caldera vs. CPU-powered columnar engines for Q6 of TPC-H. Time for Caldera includes data transfer costs.

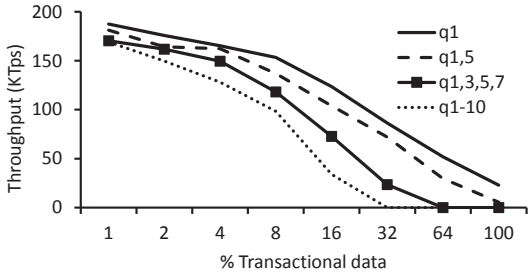


Figure 7.4 – OLTP transaction throughput in the presence of OLAP queries as we vary the OLTP working set and the degree of data freshness.

7.5 Evaluation

In this section, we present an evaluation of Caldera to show that the H^{2}TAP architecture can be implemented in practice and can offer performance competitive to that of state-of-the-art OLTP and OLAP engines. As described in Section 7.4, Caldera uses three features to tackle the challenges posed by the H^{2}TAP architecture, namely, software snapshotting for cross-archipelago HTAP, message-passing for transaction processing without CC, and PAX as the hybrid layout that enables data sharing across mixed workloads. Thus, in this section, we present the performance and scalability of these three aspects and compare Caldera with Silo [241], a main-memory OLTP engine, MonetDB [58], an open-source column store, and "DBMS-C", a commercial column store.

Experimental setup. All experiments are conducted on a server running RHEL 7.2, equipped with two 12-core Intel Xeon E5-2650L v3 CPUs, 256GB RAM, and a GeForce GTX 980 GPU with 4GB memory. Although the hardware we use supports system-wide CC, Caldera uses it only as the message passing substrate for inter-thread communication.

7.5.1 HTAP with software snapshotting

We present the OLTP throughput and OLAP response time achieved by Caldera under a mixed workload. For these experiments, we use the TPC-H (SF-300) dataset. We use TPC-H Q6, a selection over the lineitem table, as the OLAP query. In our OLTP workload, each transaction performs ten read-modify-update operations on records randomly chosen from the lineitem table. Thus, the OLTP workload is similar to an update-only YCSB workload [85] with a theta value (zipfian distribution) of zero. We run ten OLAP queries in succession on the GPU. The OLTP workload is executed by the CPU until all OLAP queries terminate. We use the snapshotting flexibility of Caldera to demonstrate the performance-freshness trade off posed by our software shadow copying implementation. Further, it is common in real-world deployments for transactions to access only a "hot" fraction of the dataset [166], whereas OLAP queries scan through all the data. We make the target key range used by the OLTP workload a parameter so that we test sensitivity to skewed OLTP working set sizes.

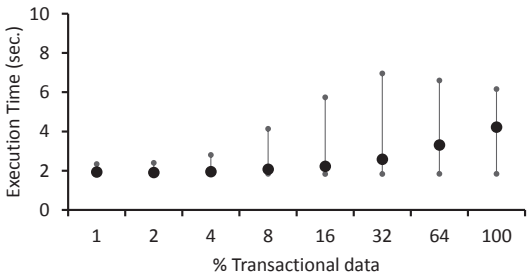


Figure 7.5 – Execution time of OLAP queries in the presence of OLTP queries. All OLAP queries share a single snapshot, but OLTP-triggered copy-on-write stresses memory bandwidth.

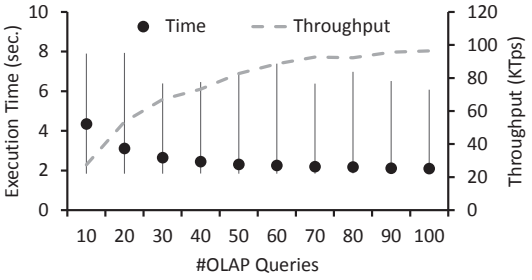


Figure 7.6 – Execution time of OLAP queries and throughput of OLTP queries. We increase the number of queries that share a snapshot from 10 to 100. Increasing snapshot sharing improves performance.

Figure 7.3 shows the OLAP query execution time under MonetDB, DBMS-C, and Caldera in the absence of transactions. Both MonetDB and DBMS-C parallelize the query across all 24 cores. MonetDB is 1.27× faster than DBMS-C because it benefits from the use of secondary indexes. Caldera exploits the massive parallelism of the GPU to provide 4.15× and 5.29× speedup over MonetDB and DBMS-C, even though the table is streamed from host memory.

Figure 7.4 shows the OLTP throughput achieved by Caldera as we vary the working set size from 1% to 100%. The four lines show the throughput as we increase data freshness by varying snapshot frequency from one across all ten OLAP queries to one per OLAP query. Clearly, transactional throughput deteriorates when we increase the working set size or the frequency of snapshots due to software overhead, as Caldera incurs the cost of performing a copy-on-write the first time data is modified after each snapshot.

Snapshotting also affects OLAP response time. Figure 7.5 shows the average, minimum, and maximum analytical query response times for Caldera when all ten queries share one snapshot as we vary the (OLTP) working set size from 1% to 100%. In the presence of snapshotting, both analytical queries running on the GPU and transactions running on CPU compete for memory bandwidth due to the memory-intensive copy-on-write process. This results in a 2× increase in average response time and a 3× increase in maximum response time. Note that this overhead is not exclusive to the shadow copy implementation of Caldera: Fork-based snapshotting implementations also suffer under update intensive workloads [248]. In addition to such snapshotting-related overheads, current HTAP engines also exhibit the house effect as transaction throughput collapses due to processor resource contention caused by interference between OLAP and OLTP workloads [210]. Under Caldera, in contrast, processor resource contention never occurs due to the strict separation of workloads provided by the archipelago abstraction. Contention for memory bandwidth is purely due to the software overhead of our copy-on-write mechanism and can be reduced using three techniques.

The first optimization is to trade off a degree of data freshness for improved performance by sharing a snapshot across several OLAP queries. Figure 7.6 shows the throughput and response

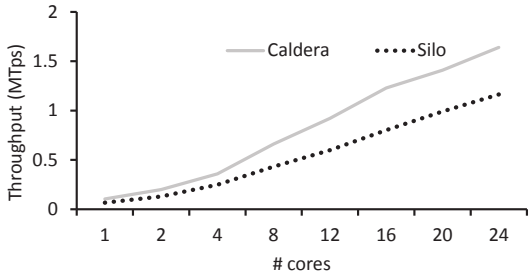


Figure 7.7 – TPC-C scalability as the number of cores increase.

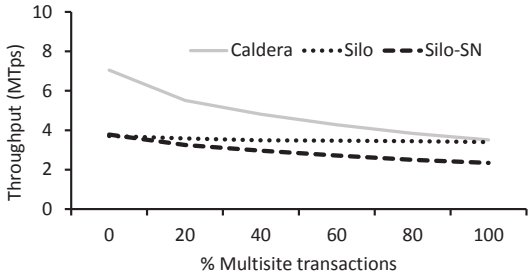


Figure 7.8 – Throughput as the percentage of multi-site transactions increases.

time for Caldera when we fix the OLTP working set to 100% – the worst case in Figure 7.5 – and vary the number of queries that share a snapshot from 10 to 100. Initially, almost all transactions perform copy-on-write. Analytical queries that are executed concurrently with these transactions suffer due to shared memory bandwidth. This explains the high worst-case response time for analytical queries. As the copy-on-write process converges, both transactional throughput and analytical response time improve substantially. Comparing Figures 7.4 and 7.6, we observe that sharing a snapshot across 100 queries provides nearly a 5× improvement in OLTP throughput even if the working set covers 100% of the data set.

Second, as shown in Figure 7.4, limiting the OLTP working set to less than 16% of the total data size limits the worst-case deterioration in throughput to only 2× even if we use one snapshot per query. Thus, hybrid data layouts that perform hot–cold data classification [161] will enable Caldera to further reduce the impact on OLTP throughput.

Third, profiling revealed that both memory allocation and memory copying performed during the shadow-copy operation were sources of overhead. Thus, optimizing shadow copying by using alternate snapshotting implementations [222, 248] is another approach for improving OLTP throughput.

7.5.2 OLTP with message passing

Next, we compare the performance and scalability of Caldera against Silo for OLTP workloads. To avoid confounding performance effects caused by memory allocation, and to keep the comparison fair, we use the NSM data layout, and also use malloc as the memory allocator for Caldera.

The first experiment investigates the scalability of both systems for the NewOrder transaction of the TPC-C benchmark. For both systems, we assign a warehouse to a thread and increase the number of threads (and hence the number of warehouses). Figure 7.7 reports throughput at various thread counts; both systems scale well. Caldera outperforms Silo due to 1) better data locality provided by partitioning, 2) better code locality due to the lack of thread synchronization, and 3) limited message passing overhead because only 10% of NewOrder transactions require remote accesses.

The next experiment investigates throughput sensitivity in the presence of multi-site transactions. We use a read-only microbenchmark in which each transaction reads ten records from a table of 24M records partitioned across 24 cores. Single-site transactions read all ten records from the local partition. Multi-site transactions read two records from a random remote partition and the remaining eight from the local partition. We compare Caldera with two deployments of Silo, namely, *Silo* and *shared-nothing Silo (SN-Silo)*. The default configuration uses a single instance of Silo over all cores. SN-Silo represents how one could use current OLTP engines on emerging non-CC multi-cores; the SN-Silo setup uses one instance of Silo per core and a distributed transaction layer to coordinate multi-site transactions using the two-phase commit (2PC) protocol.

Figure 7.8 shows the throughput achieved by all three systems as the fraction of multi-site transactions increases. Both Caldera and SN-Silo are affected by multi-site transactions, but for very different reasons; Caldera suffers due to the use of CC as the message passing mechanism while SN-Silo suffers due to the overheads of 2PC. Thus, for emerging hardware, replacing CC with hardware message passing will benefit Caldera, but not SN-Silo. Despite the message passing overhead, Caldera can match Silo's throughput, showing that the message passing-based design used by Caldera provides performance competitive with that of state-of-the-art OLTP engines.

7.5.3 Data sharing with PAX

The next experiment examines the suitability of PAX for OLAP operations executed on GPUs. For this experiment, we use a main-memory-resident 16 GB table of 270M records. Each record is comprised of 16 integer attributes. We use three different storage layouts for the table: DSM, PAX, and NSM. We set the size of the PAX page to 4KB. Each PAX page contains 16 minipages, and each minipage contains 64 values. We then launch five instances of the following query template:

```
SELECT SUM(col1 + ... + colN) FROM dataset
```

Each instance accesses 1, 2, 4, 8, or 16 attributes, respectively. Figure 7.9 depicts the response time for each instance. NSM has the slowest response times because it leads to sub-optimal data access patterns. Specifically, GPUs manage threads in groups. The ideal access pattern in the context of GPUs is one for which all threads in a group perform *coalesced accesses*, i.e., they access a contiguous chunk of memory. When executing a query over NSM data, the values for col1, col2, etc., are not stored contiguously, thus resulting in multiple expensive memory transactions.

PAX and DSM have almost identical response times, with the former being slightly slower. Both the PAX and DSM layouts lead to coalesced memory accesses. In addition, both layouts minimize unnecessary data transfers through the PCIe bus. Specifically, the maximum transfer unit (MTU) through the PCIe bus typically does not exceed 512 bytes. We carefully configure the PAX layout so that the size of each minipage is close to the MTU, and thus maximize the utilization of the PCIe bandwidth.

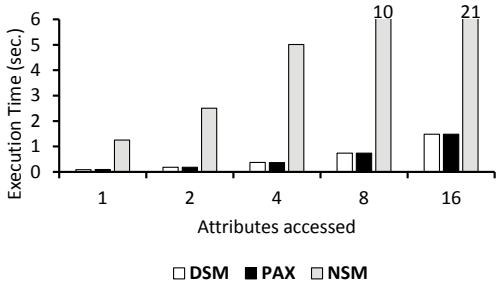


Figure 7.9 – Comparing the efficiency of different data layouts for GPU-based computations.

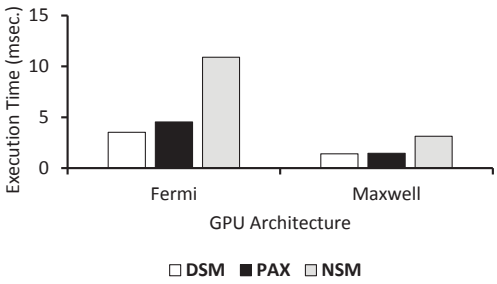


Figure 7.10 – Comparing different data layouts when all data is GPU resident.

While our previous experiment showed that NSM suffers due to its inability to perform coalesced accesses, PAX and DSM are able to effectively saturate the PCIe bandwidth. The GPU memory, however, provides an order of magnitude higher bandwidth compared to PCIe. As on-board GPU memory continues to increase in capacity, an important question is whether PAX lags behind DSM if all data were local to the GPU. To answer this question, we repeated the previous experiment while storing all data in GPU memory. Due to limited memory capacity, we reduced the dataset size from 16GB to 1GB. Figure 7.10 shows the response time for the three layouts when the query touches only two attributes out of 16. We only report the kernel execution time and not data transfer time for two GPUs belonging to different generations, namely a Fermi GPU (Tesla M2090) and the Maxwell GPU (GTX 980). There are three important observations.

First, comparing the two GPUs, we see that the Maxwell GPU provides a 2.5×, 3.1×, and 3.5× improvement for DSM, PAX, and NSM layouts respectively. These results are encouraging because despite being just a consumer-grade graphics card, the Maxwell GPU (GTX 980) outperforms a previous-generation compute accelerator (Tesla M2090).

Second, comparing relative performance of each layout within a GPU generation, we see that NSM is 3× slower than DSM on Tesla and only 2× slower on Maxwell. Similar, PAX is 1.3× slower on Tesla but matches DSM performance on Maxwell. This result is in sharp contrast with the UVA results we reported in Figure 7.9, where NSM was 13.74× slower than DSM. This shows that modern GPUs have vastly reduced the performance impact of non-coalesced memory accesses when data fits in GPU memory. Thus, using a PAX-like storage layout that acts as the middle ground between OLTP-oriented NSM and OLAP-oriented DSM is a viable option for H^2TAP. A possible next step would be crafting a new data layout dynamically depending on the workload requirements [37, 120], e.g., storing frequently accessed attributes together in a group of columns.

Overall, the results indicate that it is possible to realize H^2TAP in practice and show many of the opportunities and challenges involved in designing H^2TAP engines.

7.6 Summary

Modern database engines are designed to work on multi-socket multi-cores that provide abundant homogeneous parallelism, system-wide CC, and global shared memory. As a result, they are mismatched with emerging server hardware, which makes both parallelism and CC support heterogeneous. We introduce H^{2}TAP, a new architecture for building database engines on such hardware. Using Caldera, a prototype H^{2}TAP engine, we show that the H^{2}TAP architecture can be realized in practice and can match the performance of state-of-the-art specialized OLTP and OLAP engines.

8 The Big Picture

This thesis examines the problems that arise in modern-day analytics over heterogeneous datasets. Existing data analysis solutions typically involve transforming all datasets in a single proprietary format and loading them in a warehouse prior to initiating analysis. Such solutions do not scale with the increasing volume and diversity of data and query workloads. In addition, they are incompatible with scenarios in which data movement is prohibitive, while they are not flexible enough for users to analyze their data ad-hoc. In summary, monolithic designs with static query processing primitives are unable to efficiently support the variety of data formats, models, locations, and analysis types required by modern applications.

This thesis makes the case for *just-in-time* (JIT) databases – systems that abolish static decisions, defining optimized data access and manipulation primitives on demand at runtime – that offer data virtualization; abstracting data out of its form, and manipulating it regardless of the way it is stored or structured. JIT database systems reason about data model heterogeneity through the use of expressive languages and algebras. They also mask data location heterogeneity through the use of compilers and optimizers that operate over virtual, simplified data schemas, yet generate sophisticated query plans based on the actual involved data stores. To minimize data-to-insight time, JIT systems can operate over data in situ and employ code generation techniques to adapt their internals – both query primitives and data structures – to the underlying data, the query workload, and the available hardware.

The ultimate goal of just-in-time database systems is to decouple the type of analysis performed from the original data layout and allow users to perform their analysis across data stores, data models, and data formats, but at the same time experience the performance offered by a custom system that has been built on demand to serve their specific use case. This chapter summarizes the contributions of this thesis and discusses a number of ongoing efforts to address open challenges related to just-in-time database systems and unconditional data virtualization.

8.1 Unconditional data virtualization: What we did

Every work presented in this thesis redesigns a layer of the data analysis stack to tackle a challenge stemming from data heterogeneity, with the end goal being the design of a just-in-time database system [144]. Describing the data analysis stack bottom-up, and starting from the data access part, dynamically generated access paths [145] remove the overheads of traditional scan operators [100], mask data format heterogeneity, and enable a query engine to operate natively over diverse datases – both binary ones as well as verbose, textual ones.

In terms of query processing techniques, we introduce a two-phase process to manage heterogeneous data models without sacrificing performance [143, 144]: In the first phase, we use a rich query algebra to express analysis over different collection types, such as relations and arbitrarily nested hierarchies. In the second phase, our design allows a system to adapt its behavior and its code based on the current type of analysis. Specialization to incoming queries occurs through the use of runtime code generation. We envision that just-in-time database systems will eventually specialize themselves to the available heterogeneous hardware [42] (CPUs, GPGPUs, FPGAs, etc.) to fully exploit the resources of the modern server.

When data resides across heterogeneous data stores, users typically handcraft their analysis to explicitly consider the characterictics of the various data sources and identify optimization opportunities, rendering the overall analysis non-declarative and convoluted. As in the previous case of query processing, we introduce another two-phase process to handle query formulation and optimization across heterogeneous data stores: In the first phase, a data virtualization module uses a location-aware query compiler to rewrite analysis tasks expressed over location-agnostic data views. In the second phase, the optimizer of the virtualization module produces query plans that consider the characteristics and particularities of the data stores holding the data. Coupling the virtualization module with a query engine results in a system that provides the necessary abstractions to query datasets that are dispersed across multiple data stores, and at the same time allows for minimal response times [146].

Finally, the topmost layer of data analysis involves interpreting the returned data, and figuring out whether the returned information is meaningful. It is highly likely that the data contains inconsistencies and requires some cleaning effort before becoming useful. To this end, this thesis proposes CleanM [115], a declarative query language that exposes a wide variety of parameterizable data cleaning primitives which users can apply over their data. CleanM translates all cleaning operations to an optimizable, parallelizable calculus, and optimizes them all as one unified task.

8.2 Unconditional data virtualization: Next steps

The design of just-in-time database systems is a step in the ongoing effort towards unconditional data virtualization. Further advancement requires addressing a number of challenges, as described below:

- **"To load or not to load", and "One storage layout does not fit all".**

  Querying data in situ removes the bottleneck of the costly data loading process into a DBMS prior to launching any queries. On the other hand, analytical DBMS operate over custom binary data representations for performance reasons; they pick the data representation and layout that is more suitable for the expected type of analysis. Thus, unless "raw" data comes in a compact, binary serialization, in situ query engines typically use aggressive caching policies to leverage previous data accesses and avoid re-paying to access the same raw data subsets multiple times. As a result, after a number of queries, an in situ query engine essentially operates over a binary subset of the original dataset, which corresponds to the working set of a user.

  Even when aggressively caching raw data, it is non-trivial for a JIT query engine to decide what the layout of its caches should be to minimize query running times: In the case of heterogeneous data, it is likely that users want to launch different types of analysis (e.g., both OLAP queries and navigational queries over hierarchies) over the same data, otherwise there will be a mismatch between the type of analysis and the data. For example, purely relational data layouts are not always well-suited for efficiently querying nested data. Instead, practitioners consider nested columnar layouts [9, 182] as a more suitable option than relational row-oriented and column-oriented layouts. Besides the issue of storage layout, a JIT query engine has to decide which cached data is "most valuable" to it. Specifically, the cost of reading and parsing raw data varies widely across heterogeneous formats. JIT query engines typically evict elements from their caches using cost-oblivious algorithms, such as LRU [36, 145, 197], ignoring that an element to be evicted can be very expensive to reconstruct (e.g., because the engine read it from a deeply nested JSON file).

  Addressing the issues of dynamic data layouts and caching in the context of heterogeneous data management requires careful monitoring of the workload and awareness of the cost paid to populate data caches. Our current work focuses on a cost-based cache manager that uses timing measurements and workload monitoring to automate decisions about caching policy [45]. Using workload monitoring information, the cache manager automatically switches to the best performing in-memory layout for caching nested data. Timing measurements further enable the cache manager to make more informed cache eviction decisions than LRU. Finally, the cache manager avoids high caching overhead by choosing dynamically between a low and high overhead caching scheme for previously unseen queries.

- **Indexing over raw data.** This thesis focuses on analytical use cases, and treats value-based indexes as an orthogonal, optional optimization [48, 148]. For file types that incorporate indexes over their content, the generated access paths traverse the indexes to speed up accesses. In the rest of the cases, instead of populating an index, the generated access paths opt to build binary columnar caches, which also reduce the data footprint of subsequent queries accessing a cached column. Still, there are scenarios in which avoiding full scans of tabular files with a row-oriented data layout accelerates analytical queries [197].

  It is both straightorward and beneficial to couple a just-in-time query engine with indexing

support. Depending on the workload, the analytical scenarios that this thesis examines can benefit from both lightweight, value-existence indexes, (e.g., zone maps), as well as value-position indexes (e.g., B^+-Tree). A prominent example comes from Slalom [197], an in situ query engine that makes on-the-fly partitioning and indexing decisions based on information collected from the underlying raw data. A just-in-time query engine, coupled with the partitioning and indexing tuner of Slalom, can adaptively tune indexing structures to adapt to the query workload even better.

- **Handling raw data updates.** Accessing data in situ is one of the main motivations of this thesis. It has been proposed in the context of analytical processing, where in-place data updates are infrequent; when updates do occur, they typically involve users directly extending one of the data sources / files, or simply adding a new source / file to the system. In data-append scenarios, updating any existing auxiliary structures – positional map indexes and data caches – involves extending them on the first access to the "fresh" pieces of the data. As for the case of adding a new file, no auxiliary data structures have been created for the file in the past, thus no particular actions are required.

  Proteus currently targets read-only and append-like analytical workloads. In case of append-like updates, Proteus extends its existing auxiliary structures (e.g., caches). For the case of finer-grained updates, a promising step is incorporating the update scheme of Alpine [41]: Alpine proposes a logical partitioning scheme over the input data files. Specifically, for each partition, Alpine stores an identifier that is sufficient to identify the existence of an in-place update within the partition. The identifier comprises an MD5 hash code of the contents within that partition, the starting and ending binary positions of the partition in the file, as well as the characters corresponding to those positions. When the Alpine engine detects an update, it updates the positional map and any data caches corresponding to the partition.

- **Vectorization vs. compilation.** Query compilation facilitates the generation of a query implementation that maximizes pipelining. The generated code keeps data in CPU registers as much as possible and operates over tuples in tight loops with high instruction locality. Still, pipelined query engines are not the de facto best-performing execution engines in every scenario; there are queries types for which, depending on the involved operators and their selectivities [215], a vectorized columnar executor can outperform a pipelined one [232, 202].

  In the context of accessing heterogeneous data, pipelined query processing has an added benefit: data does not have to reside in columns before query execution can start. In other words, a vectorized executor pays a materialization cost both to create columns from the underlying heterogeneous data and to create the intermediate results between query operators. Still, there are use cases in which the initial materialized columns can be subsequently treated as data caches, and thus compensate for the effort spent in populating them. In summary, it will be beneficial for a just-in-time query engine to be able to generate both fully pipelined code, as well as vectorized code on a per query case [228] – or couple pipelined and vectorized code for a single query implementation [161]; such flexibility has

the potential to maximize the JIT query engine's degree of adaptivity to the characteristics of each incoming query.

- **Reducing compilation overhead.** Generating and compiling the code corresponding to a query at runtime introduces an overhead in the total query evaluation time. The overhead varies depending on the code generation infrastructure used: Typically, generating an "external" C/C++ library per query introduces an overhead of a few seconds [190, 231, 145]. On the other hand, generating the LLVM intermediate representation typically takes 10s-100s of milliseconds. Still, if one wants to use a just-in-time query engine for short-running tasks (e.g., OLTP workloads), the compilation cost can become significant.

  Besides relying on low-level compiler infrastructure, a system can further reduce the compilation cost by caching the code generated by previous queries; the same cached code can be re-used in subsequent occurrences of similar queries. If code re-use is a major requirement, it is also possible for a system to generate code that is "parameterizable". Specifically, instead of compiling the entire query logic, the system's code generator can leave out some information, such as the value of an integer constant used in a filtering expression. Then, the same code template can be used by all queries that adhere to the same template but have different values for said filtering expression. Obviously, leaving out information from the generated code facilitates reuse, but can also hurt performance if the generated code ends up having to "interpret" significant parts of the query plan.

- **Query optimization and scheduling across heterogeneous processors.** Transactions require synchronization at multiple levels (concurrency control protocols at the logical level, latching at the physical level, atomics at the hardware level). Therefore, the H^2TAP work restricts the membership of task-parallel archipelagos to CPUs. On the other hand, OLAP queries can be parallelized well on both CPUs and GPUs, therefore the data-parallel archipelago benefits from being heterogeneous. Given the processor heterogeneity, a given OLAP query could potentially be executed on just CPUs, just GPUs, or a mix of both [61, 127, 142]. Thus, an important topic that requires further research is query optimization and scheduling in the heterogeneous OLAP archipelago.

  GDB [127] is one of the first prototypes to investigate extensions to analytical cost models in the CPU–GPU query coprocessing scenario for deciding optimal operator placement. CoGaDB [61] is a more recent effort that uses cost models based on observed query execution time that are learned on-the-fly and continuously refined for both picking an optimal query plan and the placement of operators across CPUs and multiple GPUs. We plan to extend Caldera with such heterogeneity-aware query optimizers in the future.

- **Utilizing the full range of data-parallel hardware.** Over the past few years, processor vendors have introduced several new heterogeneous hardware accelerators that compete with GPUs for accelerating data-parallel workloads. For instance, the Intel Many Integrated Core processor (also known as Xeon Phi) packs several hyperthreaded, low-frequency in-order cores together with high-bandwidth memory in a single package to provide an order-of-magnitude more hardware contexts than server-grade Xeon processors. The Intel HARP

platform integrates Field Programmable Gate Arrays (FPGAs) and Xeon processors in a single multi-socket system. Recent research has shown that analytical workloads benefit from such heterogeneous hardware [137, 184].

While the Caldera prototype focuses on GPUs, the H^{2}TAP architecture is independent of the type of data-parallel hardware used for accelerating OLAP queries. In fact, given that the H^{2}TAP architecture decouples cache coherence from shared memory, it can take advantage of the simpler data-parallel hardware that does not necessarily support system-wide cache coherence. Further, the use of query compilation makes the overall architecture hardware-agnostic; any processor can be integrated into the Caldera framework as long as the query compiler generates specialized code for the target processor.

A

A.1 Spam Analysis Queries

Due to legal restrictions, we are not allowed to disclose the exact Symantec real-world workload presented in Section 4.7.2. Nevertheless, the current section presents some indicative SQL queries used, for which table and field names have been anonymized.

For presentation purposes, when dealing with JSON data, we use the PostgreSQL JSON extensions [208]. In the FROM clause of following SQL queries, "bin", "json" and "csv" indicates that the query operates over binary tabular data, JSON data, and comma-separated-values data, respectively.

Queries over binary data.

```
SELECT MAX(w), MAX(x), SUM(y), SUM(z), COUNT(*)
FROM bin
WHERE f1 > val1 AND f2 < val2 AND f3 < val3
      AND f4 > val4 AND f5 <= val5
GROUP BY z;
```

Queries over CSV data.

```
SELECT f5, MAX(a), MAX(b), COUNT(*)
FROM csv
WHERE f1 > val1 AND f2 < val2 AND f3 = 'foo'
      AND f4 < val3 AND (f5 = 'xx' OR f5 = 'yy' OR f5 = 'zz')
GROUP BY f5;
```

Queries over JSON data.

PostgreSQL treats JSON as an explicit data type, therefore field manipulation of a JSON object requires overloaded constructs. In the following query, $(obj->>'x')::int$ accesses field x of a JSON object, and treats it as an integer.

```
SELECT (obj->'z'->>'z1')::int,
        MAX((obj->>'x')::int), count(*)
FROM json
WHERE (obj->>'x')::int < val1 AND
      (obj->>'y')::int > val2
GROUP BY (obj->'z'->>'z1')::int;
```

Unnesting a JSON array using PostgreSQL involves a call to the *json_array_elements* function, as well as a nested query to continue manipulation of the results.

```
SELECT MAX(x2), COUNT(*)
FROM (SELECT (obj->>'x')::int as x2,
             json_array_elements((obj->>'y')::json)
      FROM json
      WHERE (obj->>'a')::int < val1) internal;
```

Queries over a combination of datasets.

```
SELECT csv.f4, MAX(x), SUM(y), COUNT(z)
FROM  csv JOIN json ON (csv.f = (obj->>'f')::int)
WHERE csv.f1 > val1 AND csv.f2 < val2 AND
      csv.f3 = 'foo' AND
      (csv.f4 < val3 OR (obj->>'f5')::int > val4)
      AND (obj->>'f6')::int < val5
      AND (obj->'f7'->>'g')::int = val6
GROUP BY csv.f4;
```

```
SELECT bin.f10, MAX(y), COUNT(*)
FROM bin
JOIN csv  ON (bin.f = csv.f)
JOIN json ON (bin.f = (obj->>'f')::int)
WHERE bin.f1 > val1 AND bin.f2 < val2 AND
      bin.f3 < val3 AND bin.f4 > val4 AND
      bin.f5 < val5 AND csv.f6 > val6 AND
      csv.f7 < val7 AND (obj->>'f8')::int > val8
      AND (obj->>'f9')::int < val9
GROUP BY bin.f10;
```

Bibliography

[1] Apache Avro. https://avro.apache.org/. 5.3

[2] Apache Calcite project. https://calcite.apache.org. 5.2, 5.8.2

[3] Apache Cassandra. http://cassandra.apache.org. 5.6.1, 5.7

[4] Apache Cassandra. Partitioners. https://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architecturePartitionerAbout_c.html. 5.6.1

[5] Apache Drill. https://drill.apache.org/. 2.1.3, 4.2

[6] Apache Flink. https://flink.apache.org. 5.3

[7] Apache Flume. http://flume.apache.org. 5.2

[8] Apache Hadoop. https://hadoop.apache.org/. 2.1.3, 5.3

[9] Apache Parquet. https://parquet.apache.org/. 4.2, 5.3, 5.5.1, 8.2

[10] Apache Sqoop. http://sqoop.apache.org. 5.2

[11] Google Gson. https://github.com/google/gson. 4.5.2

[12] Jackson. https://github.com/FasterXML/jackson. 4.5.2

[13] KNIME. https://www.knime.org/. 2.5.2

[14] LLVM's Analysis and Transform Passes. http://llvm.org/docs/Passes.html. 4.5.1

[15] MapD. https://www.mapd.com/. 7.2

[16] Paxata. https://www.paxata.com/. 2.5.2

[17] Pentaho. http://www.pentaho.com/. 2.5.2

[18] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org. 3.5.3

[19] RapidJSON. http://rapidjson.org/. 4.5.2

[20] TPCx-BB. http://www.tpc.org/tpcx-bb. 5.1, 5.7.1

[21] Apache Arrow, 2017. http://arrow.apache.org. 5.2, 5.8.2

[22] G. Aad et al. The ATLAS Experiment at the CERN Large Hadron Collider. *Journal of Instrumentation*, 3(8):1–438, 2008. 3.6

[23] C. L. Abad, N. Roberts, Y. Lu, and R. H. Campbell. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization, IISWC '12*, pages 100–109, 2012. 5.1

[24] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013. 3.3

[25] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *Proceedings of the 32nd IEEE International Conference on Data Engineering, ICDE '16*, pages 1134–1145, 2016. 2.5, 2.5.1, 2.5.2

[26] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999. 2.2

[27] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Journal on Digital Libraries*, 1(1):68–88, 1997. 2.2

[28] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009. 4.2, 5.2

[29] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings*, pages 1–10, 2013. 1.1, 2.1.3

[30] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD '96*, pages 137–148, 1996. 4.2

[31] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD '96*, pages 137–148, 1996. 5.6.1

[32] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00*, pages 496–505, 2000. 4.6

[33] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of 27th International Conference on Very Large Data Bases, VLDB '01*, pages 169–180, 2001. 7.3, 7.4

[34] A. Ailamaki, V. Kantere, and D. Dash. Managing scientific data. *Communications of ACM*, 53(6):68–78, 2010. 1

[35] M. Al-Kateb, P. Sinclair, A. Crolotte, L. Ma, G. Au, and S. Nair. Optimizing UNION ALL join queries in teradata. pages 1209–1212, 2017. 5.1, 5.2, 5.6.1

[36] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, pages 241–252, 2012. 1.1, 2.1.1, 2.1.3, 3.1, 3.2, 3.4.1, 3.4.2, 3.4.2, 4.5.2, 8.2

[37] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 1103–1114, 2014. 2.3, 7.3, 7.4, 7.5.3

[38] A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014. 2.1.3, 5.3

[39] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014. 2.1.3, 4.2

[40] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *Proceedings of the VLDB Endowment*, 9(3):120–131, 2015. 2.5.2

[41] A. Anagnostou, M. Olma, and A. Ailamaki. Alpine: Efficient In-Situ Data Exploration in the Presence of Updates. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, SIGMOD '17*, pages 1651–1654, 2017. 2.1.3, 8.2

[42] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. The Case For Heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research, CIDR '17*, 2017. 8.1

[43] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, pages 1383–1394, 2015. 2.1.3, 2.5.2, 4.1, 4.2, 5.1, 5.2, 5.3, 5.3, 6.1, 6.5, 6.6

[44] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD '16*, pages 583–598, 2016. 7.3, 7.4

[45] T. Azim, M. Karpathiotakis, and A. Ailamaki. ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data. *Proceedings of the VLDB Endowment*, 2018. To Appear. 8.2

[46] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013. 4.5.1

[47] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the 10th European Conference on Computer Systems, EuroSys '15*, 2015. 2.4.2

[48] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M. Kim, O. Koeth, J. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business Analytics in (a) Blink. *IEEE Data Engineering Bulletin*, 35(1):9–14, 2012. 8.2

[49] A. Baumann, P. Barham, P. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP '09*, pages 29–44, 2009. 2.4.2, 7.1

[50] A. Baumann, C. Hawblitzel, K. Kourtis, T. Harris, and T. Roscoe. Cosh: Clear OS Data Sharing In An Incoherent World. In *2014 Conference on Timely Results in Operating Systems, TRIOS '14*, 2014. 2.4.2, 7.1

[51] L. Berti-Equille, T. Dasu, and D. Srivastava. Discovery of complex glitch patterns: A novel approach to Quantitative Data Cleaning. In *Proceedings of the 27th IEEE International Conference on Data Engineering, ICDE '11*, pages 733–744, 2011. 2.5.2

[52] L. Berti-Équille, J. M. Loh, and T. Dasu. A masking index for quantifying hidden glitches. *Knowledge and Information Systems*, 44(2):253–277, 2015. 2.5.2

[53] K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, R. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. C. Truong, B. V. der Linden, B. Vickery, and C. Zhang. System RX: one part relational, one part XML. In *Proceedings of the 2005 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '05, pages 347–358, 2005. 4.2

[54] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *Proceedings of the VLDB Endowment*, 4(12):1272–1283, 2011. 2.1.3, 4.2

[55] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 385–396, 2014. 2.1.3

[56] J. Bleiholder and F. Naumann. Declarative Data Fusion – Syntax, Semantics, and Implementation. In *Proceedings of the 9th East European Conference on Advances in Databases and Information Systems, ADBIS '05*, 2005. 2.5.2

[57] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06*, pages 479–490, 2006. 4.2

[58] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of ACM*, 51(12):77–85, 2008. 2.1.1, 2.1.3, 2.3, 3.3, 4.5, 7.5

[59] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *2nd Biennial Conference on Innovative Data Systems Research, CIDR '05*, pages 225–237, 2005. 2.1.2, 2.3, 3.1, 3.3, 3.5.1

[60] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *Proceedings of the VLDB Endowment*, 7(13), 2014. 6.2.2

[61] S. Breß, H. Funke, and J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD '16*, pages 1891–1906, 2016. 8.2

[62] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 15:1–35, 2014. 2.4.1

[63] R. Brun and F. Rademakers. ROOT - An Object Oriented Data Analysis Framework. In *Proceedings of the Workshop on Artificial Intelligence in High Energy and Nuclear Research, AIHENP '96*, 1997. 3.1, 3.6

[64] R. Brunel, J. Finis, G. Franz, N. May, A. Kemper, T. Neumann, and F. Färber. Supporting hierarchical data in SAP HANA. In *Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE '15*, pages 1280–1291, 2015. 4.2

[65] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, and I. Manolescu. Invisible Glue: Scalable Self-Tuning Multi-Stores. In *7th Biennial Conference on Innovative Data Systems Research, CIDR '15*, 2015. 1, 4.1, 4.2, 5.2

[66] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large XML repositories. In *Proceedings of the 21st IEEE International Conference on Data Engineering, ICDE '05*, pages 261–272, 2005. 4.2

[67] P. Buneman, S. B. Davidson, K. Hart, G. C. Overton, and L. Wong. A Data Transformation System for Biological Data Sources. In *Proceedings of the 21st International Conference on Very Large Data Bases, VLDB '95*, pages 158–169, 1995. 2.2

[68] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000. 2.2

[69] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. pages 87–96, 1994. 2.2, 2.2

[70] J. Cai and A. Shrivastava. Software Coherence Management on Non-coherent Cache Multi-cores. In *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems, VLSID '16*, pages 397–402, 2016. 2.4.2

[71] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. T. II, J. H. Williams, and E. L. Wimmers. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering - Distributed Object Management, RIDE-DOM '95*, pages 124–131, 1995. 4.2, 5.2, 5.6.1

[72] M. J. Carey and H. Lu. Load Balancing in a Locally Distributed Database System. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, SIGMOD '1986*, pages 108–119, 1986. 5.2

[73] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A History and Evaluation of System R. *Communications of ACM*, 24(10):632–646, 1981. 2.3

[74] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON document stores in relational systems. In *Proceedings of the 16th International Workshop on the Web and Databases, WebDB 13*, pages 1–6, 2013. 4.2

[75] S. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Information Processing Society of Japan, IPSJ '94*, pages 7–18, 1994. 4.2, 5.2

[76] L. J. Chen, P. A. Bernstein, P. Carlin, D. Filipovic, M. Rys, N. Shamgunov, J. F. Terwilliger, M. Todic, S. Tomasevic, and D. Tomic. Mapping XML to a wide sparse table. In *Proceedings of the 28th IEEE International Conference on Data Engineering, ICDE '12*, 2012. 4.2

[77] L. J. Chen, P. A. Bernstein, P. Carlin, D. Filipovic, M. Rys, N. Shamgunov, J. F. Terwilliger, M. Todic, S. Tomasevic, and D. Tomic. Mapping XML to a wide sparse table. *IEEE Transactions on Knowledge and Data Engineering, TKDE '14*, 26(6):1400–1414, 2014. 4.2

[78] Y. Cheng and F. Rusu. Parallel In-Situ Data Processing with Speculative Loading. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 1287–1298, 2014. 1.1, 2.1.3

[79] Y. Cheng and F. Rusu. SCANRAW: A database meta-operator for parallel in-situ processing and loading. *ACM Transactions on Database Systems, TODS '15*, 40(3):19:1–19:45, 2015. 2.1.3

[80] Y. Cheng, W. Zhao, and F. Rusu. Bi-Level Online Aggregation on Raw Data. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, SSDBM '17*, pages 10:1–10:12, 2017. 2.1.3

[81] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, Prabhat, and R. D. Ryne. Parallel index and query for large scale data analysis. In *Conference on High Performance Computing Networking, Storage and Analysis, SC '11*, pages 30:1–30:11, 2011. 2.1.3

[82] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE '13*, pages 458–469, 2013. 2.5.2

[83] X. Chu, M. Ouzzani, J. Morcos, I. F. Ilyas, P. Papotti, N. Tang, and Y. Ye. KATARA: Reliable Data Cleaning with Knowledge Bases and Crowdsourcing. *Proceedings of the VLDB Endowment*, 8(12):1952–1955, 2015. 2.5

[84] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009. 2.5.2

[85] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, pages 143–154, 2010. 7.5.1

[86] G. P. Copeland and S. N. Khoshafian. A Decomposition Storage Model. *ACM SIGMOD Record*, 14(4):268–279, 1985. 7.3

[87] W. Corno, F. Corcoglioniti, I. Celino, and E. D. Valle. Exposing Heterogeneous Data Sources as SPARQL Endpoints through an Object-Oriented Abstraction. In *Proceedings of the 3rd Asian Semantic Web Conference, ASWC '08*, pages 434–448, 2008. 2.2

[88] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, 2014. 2.3

[89] P. Cudré-Mauroux, E. Wu, and S. Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *4th Biennial Conference on Innovative Data Systems Research, CIDR '09*, 2009. 2.2

[90] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: A Commodity Data Cleaning System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, pages 541–552, 2013. 2.5.2, 6.1

[91] M. Dashti. Program Analysis and Compilation Techniques for Speeding up Transactional Database Workloads, 2017. PhD thesis, EPFL. 2.3

[92] T. Dasu and J. M. Loh. Statistical Distortion: Consequences of Data Cleaning. *Proceedings of the VLDB Endowment*, 5(11):1674–1683, 2012. 2.5.2

[93] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM*, 51(1):107–113, 2008. 2.1.3

[94] A. Deshpande and J. M. Hellerstein. Decoupled Query Optimization for Federated Database Systems. In *Proceedings of the 18th IEEE International Conference on Data Engineering, ICDE '02*, pages 716–727, 2002. 5.2

[95] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split Query Processing in Polybase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13*, pages 1255–1266, 2013. 2.1.3, 4.2, 5.2

[96] G. F. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili. Relational algorithms for multi-bulk-synchronous processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13*, pages 301–302, 2013. 2.4.1, 7.4

[97] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012. 1

[98] W. Du, R. Krishnamurthy, and M.-C. Shan. Query Optimization in a Heterogeneous DBMS. In *Proceedings of the 18th International Conference on Very Large Data Bases, VLDB '92*, pages 277–291, 1992. 5.6.1

[99] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The BigDAWG Polystore System. *ACM SIGMOD Record*, 44(3), 2015. 1, 4.1, 4.2, 5.2

[100] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki. DBMS Data Loading: An Analysis on Modern Hardware. In *Data Management on New Hardware - 7th International Workshop on Accelerating Data Analysis and Data Management*

Systems Using Modern Processor and Storage Architectures, ADMS '16 and 4th International Workshop on In-Memory Data Management and Analytics, IMDM '16, pages 95–117, 2016. 2.1.1, 8.1

[101] ESRI. Shapefile Technical Description. http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf. 3.4.1

[102] W. Fan. Data quality: From theory to practice. *ACM SIGMOD Record*, 44(3):7–18, 2015. 2.5.1, 2.5.2

[103] L. Fegaras. Incremental query processing on big data streams. *IEEE Transactions on Knowledge and Data Engineering, TKDE '16*, 28(11):2998–3012, 2016. 6.2.2

[104] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, SIGMOD '95*, 1995. 2.2

[105] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000. 2.2, 2.2, 4.3, 6.1, 6.2.2, 6.3.1, 6.3.2, 6.4

[106] M. F. Fernández, J. Siméon, and P. Wadler. An Algebra for XML Query. In *Foundations of Software Technology and Theoretical Computer Science, 20th Conference, FST TCS '00*, pages 11–45, 2000. 2.2, 4.3

[107] S. J. Finkelstein. Common Subexpression Analysis in Database Applications. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data, SIGMOD '82*, pages 235–245, 1982. 4.6

[108] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. *Proceedings of the VLDB Endowment*, 7(12):1295–1306, 2014. 5.6.1

[109] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999. 4.2

[110] H. Galhardas. Data Cleaning and Transformation Using the AJAX Framework. In *International Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE '05*, pages 327–343, 2005. 2.5.2, 6.1

[111] H. Galhardas, D. Florescu, D. E. Shasha, E. Simon, and C. Saita. Improving data cleaning quality using a data lineage facility. In *Proceedings of the 3rd International Workshop on Design and Management of Data Warehouses, DMDW '01*, page 3, 2001. 6.6

[112] M. N. Garofalakis and Y. E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97*, 1997. 5.2

[113] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's all folks! LLUNATIC goes open source. *Proceedings of the VLDB Endowment*, 7(13):1565–1568, 2014. 2.5

[114] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. Jacobsen. BigBench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13*, pages 1197–1208, 2013. 5.1, 5.7.1

[115] S. Giannakopoulou, M. Karpathiotakis, B. Gaidioz, and A. Ailamaki. An Optimizable Query Language for Unified Scale-Out Data Cleaning. volume 10, pages 1466–1477, 2017. 8.1

[116] Google. Supersonic Library. `https://code.google.com/p/supersonic/`. 3.3

[117] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the 9th IEEE International Conference on Data Engineering, ICDE '93*, pages 209–218, 1993. 2.1.1, 2.3, 4.5

[118] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *ACM SIGMOD Record*, 34(4):34–41, 2005. 1

[119] S. C. Gray, F. Özcan, H. Pereyra, B. van der Linden, and A. Zubiri. SQL-on-Hadoop without compromise. Technical report, IBM, 03 2015. 5.1, 5.4

[120] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - A Main Memory Hybrid Storage Engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010. 4.6, 7.3, 7.4, 7.5.3

[121] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '06*, 26:10–24, 2006. 2.4.2

[122] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. *Proceedings of the VLDB Endowment*, 8(12):2004–2007, 2015. 2.5.2

[123] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, SIGMOD '89*, pages 377–388, 1989. 2.1.3

[124] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97*, pages 276–285, 1997. 5.2, 5.6.1

[125] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001. 4.6

[126] B. Haynes, A. Cheung, and M. Balazinska. PipeGen: Data Pipe Generator for Hybrid Analytics. In *Proceedings of the 7th ACM Symposium on Cloud Computing, SoCC '16*, pages 470–483, 2016. 5.2

[127] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems, TODS '09*, 34(4):21:1–21:39, 2009. 2.4.1, 7.2, 7.4, 8.2

[128] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013. 2.4.1, 7.2

[129] T. Heinis, M. Branco, I. Alagiannis, R. Borovica, F. Tauheed, and A. Ailamaki. Challenges and Opportunities in Self-Managing Scientific Databases. *IEEE Data Engineering Bulletin*, 34(4):44–52, 2011. 1

[130] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993. 5.2

[131] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *IEEE International Solid-State Circuits Conference, ISSCC '10*, pages 108–109, 2010. 2.4.2, 7.1

[132] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *5th Biennial Conference on Innovative Data Systems Research, CIDR '11*, pages 57–68, 2011. 2.1.3

[133] C. G. III, F. Sironi, M. F. Kaashoek, and N. Zeldovich. Hare: a file system for non-cache-coherent multicores. In *Proceedings of the 10th European Conference on Computer Systems, EuroSys '15*, pages 30:1–30:16, 2015. 2.4.2

[134] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends® in Databases*, 5(4):281–393, 2015. 2.5, 6.2.1

[135] M. Ivanova, M. Kersten, and S. Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management, SSDBM '12*, pages 485–494, 2012. 1.1, 2.1.3

[136] M. Ivanova, M. L. Kersten, N. J. Nes, and R. Goncalves. An architecture for recycling intermediates in a column-store. volume 35, pages 24:1–24:43, 2010. 3.5.1, 4.6

[137] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proceedings of the VLDB Endowment*, 8(6):642–653, 2015. 8.2

[138] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String Similarity Joins: An Experimental Evaluation. *Proceedings of the VLDB Endowment*, 7(8):625–636, 2014. 6.1, 6.3.1, 6.3.2

[139] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the International Conference on Human Factors in Computing Systems, CHI '11*, pages 3363–3372, 2011. 2.5, 2.5.2, 6.1

[140] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15*, pages 158–169, 2015. 5.6.2, 5.8.2

[141] Y. Kargin, M. L. Kersten, S. Manegold, and H. Pirk. The DBMS - your big data sommelier. In *Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE '15*, 2015. 1.1, 2.1.3

[142] T. Karnagel, D. Habich, and W. Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *Proceedings of the VLDB Endowment*, 10(7):733–744, 2017. 8.2

[143] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *Proceedings of the VLDB Endowment*, 9(12):972–983, 2016. 6.2.2, 6.4, 7.3, 8.1

[144] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *7th Biennial Conference on Innovative Data Systems Research, CIDR '15*, 2015. 1.1, 2.1.3, 4.2, 5.1, 6.2.2, 7.3, 8.1

[145] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *Proceedings of the VLDB Endowment*, 7(12):1119–1130, 2014. 1.1, 2.1.3, 7.3, 8.1, 8.2

[146] M. Karpathiotakis, A. Floratou, F. Özcan, and A. Ailamaki. No data left behind: Real-time insights from a complex data ecosystem. *Proceedings of the 8th ACM Symposium on Cloud Computing, SoCC '17*, 2017. 8.1

[147] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th IEEE International Conference on Data Engineering, ICDE '11*, pages 195–206, 2011. 7.3

[148] M. S. Kester, M. Athanassoulis, and S. Idreos. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, SIGMOD '17*, pages 715–730, 2017. 8.2

[149] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and S. Yin. BigDansing: A System for Big Data Cleansing. In *Proceedings of the 2015*

ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pages 1215–1230, 2015. 2.5, 2.5.2, 6.1, 6.7, 6.7.3

[150] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, 1982. 6.3.1

[151] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling, 2nd Edition*. Wiley, 2002. 5.8.1

[152] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *AAAI Spring Symposium on Information Gathering*, pages 233–246, 1995. 5.2

[153] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. *Proceedings of the VLDB Endowment*, 7(10):853–864, 2014. 2.3, 4.2, 7.3

[154] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient Deduplication with Hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012. 2.5.2

[155] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for hadoop. In *7th Biennial Conference on Innovative Data Systems Research, CIDR '15*, 2015. 5.5.1, 5.6.2

[156] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000. 5.2, 5.6, 5.6.1

[157] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, SIGMOD '99*, pages 371–382, 1999. 4.6

[158] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06*, pages 802–803, 2006. 2.5.1

[159] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Proceedings of the 26th IEEE International Conference on Data Engineering, ICDE '10*, pages 613–624, 2010. 2.3, 3.1, 4.2, 4.5, 6.6, 7.3

[160] D. Laney. 3D Data Management: Controlling Data Volume, Velocity, and Variety. Technical report, META Group, February 2001. 3.1

[161] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD '16*, pages 311–326, 2016. 7.5.1, 8.2

[162] C. Lattner and V. S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization, CGO '04*, pages 75–88, 2004. 2.3, 3.4.2, 4.2, 4.5.1

[163] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: A Coherent Shared Memory Interface for Cell Be. In *17th International Conference on Parallel Architecture and Compilation Techniques, PACT '08*, pages 303–314, 2008. 2.4.2

[164] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: souping up big data query processing with a multistore system. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 1591–1602, 2014. 5.2

[165] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '02*, pages 233–246, 2002. 5.2

[166] J. J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE '13*, pages 26–37, 2013. 7.5.1

[167] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the 5th ACM Symposium on Cloud Computing, SoCC '14*, pages 6:1–6:15, 2014. 5.5.1

[168] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A fast JSON parser for data analytics. *Proceedings of the VLDB Endowment*, 10(10):1118–1129, 2017. 4.5.2

[169] F. X. Lin, Z. Wang, and L. Zhong. K2: A Mobile Operating System for Heterogeneous Coherence Domains. *ACM Transactions on Computer Systems, TOCS '15*, 33(2):4:1–4:27, 2015. 2.4.2

[170] L. Liu and M. T. Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009. 5.2

[171] Z. H. Liu, B. C. Hammerschmidt, and D. McMahon. JSON data management: supporting schema-less development in RDBMS. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 1247–1258, 2014. 4.2

[172] S. Lohr. For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights, The New York Times, 2014. 1.2, 6.1

[173] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proceedings of the 12th International Conference on Very Large Data Bases, VLDB '86*, pages 149–159, 1986. 5.6.1

[174] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering, TKDE '02*, 14(4):709–730, 2002. 4.5.1

[175] H. Markram, K. Meier, T. Lippert, S. Grillner, R. S. Frackowiak, S. Dehaene, A. Knoll, H. Sompolinsky, K. Verstreken, J. DeFelipe, S. Grant, J. Changeux, and A. Saria. Introducing the Human Brain Project. In *Proceedings of the 2nd European Future Technologies Conference and Exhibition, FET '11*, pages 39–42, 2011. 1.1

[176] M. Martin, M. Hill, and D. Sorin. Why on-chip cache coherence is here to stay. *Communications of ACM*, 55(7):78–89, 2012. 2.4.2

[177] T. G. Mattson, V. Gadepally, Z. She, A. Dziedzic, and J. Parkhurst. Demonstrating the BigDAWG Polystore System for Ocean Metagenomics Analysis. In *8th Biennial Conference on Innovative Data Systems Research, CIDR '17*, 2017. 5.2

[178] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core Network-on-a-chip Terascale Processor. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC '08*, page 38, 2008. 2.4.2, 7.1

[179] A. McCallum, K. Nigam, and L. H. Ungar. Efficient Clustering of High-dimensional Data Sets with Application to Reference Matching. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, SIGKDD '00*, pages 169–178, 2000. 6.3.2

[180] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997. 2.2

[181] E. Meijer. The world according to LINQ. *Communications of ACM*, 54(10):45–51, 2011. 2.2

[182] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment*, 3(1):330–339, 2010. 2.1.3, 4.2, 8.2

[183] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *44th International Conference on Parallel Processing, ICPP '15*, pages 739–748, 2015. 2.4.2

[184] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009. 8.2

[185] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant Loading for Main Memory Databases. volume 6, pages 1702–1713, 2013. 2.1.3

[186] R. Murthy, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, and V. Krishnamurthy. Towards an enterprise XML

architecture. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD '05*, pages 953–957, 2005. 4.2

[187] MySQL. Chapter 24. Writing a Custom Storage Engine. http://dev.mysql.com/doc/internals/en/custom-engine.html. 2.1.2

[188] F. Nagel, G. M. Bierman, and S. D. Viglas. Code Generation for Efficient Query Processing in Managed Runtimes. *Proceedings of the VLDB Endowment*, 7(12):1095–1106, 2014. 2.3

[189] F. Nagel, P. A. Boncz, and S. Viglas. Recycling in pipelined query evaluation. In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE '13*, pages 338–349, 2013. 3.5.1, 4.6

[190] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011. 2.3, 3.4.1, 3.4.2, 4.2, 4.5, 7.3, 8.2

[191] M. Nowak, K. Nienartowicz, A. Valassi, M. Lubeck, and D. Geppert. Objectivity data migration. *Proceedings of the Conference for Computing in High Energy and Nuclear Physics*, 2003. 1

[192] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide. 2.4.1

[193] NVIDIA. NVLink High-Speed Interconnect. http://www.nvidia.com/object/nvlink.html. 2.4.1

[194] NVIDIA. Parallel Thread Execution ISA Version 4.3. http://docs.nvidia.com/cuda/parallel-thread-execution. 7.3

[195] NVIDIA. Summit and Sierra Supercomputers: An Inside Look at the U.S. Department of Energy's New Pre-Exascale Systems. Technical report, 11 2014. 2.4.1

[196] A. Okcan and M. Riedewald. Processing Theta-joins Using MapReduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11*, pages 949–960, 2011. 6.5

[197] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing. *Proceedings of the VLDB Endowment*, 10(10):1106–1117, 2017. 2.1.3, 8.2

[198] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08*, pages 1099–1110, 2008. 2.1.3, 4.2, 5.1, 5.2

[199] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR*, abs/1405.3631, 2014. 4.2

[200] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI '15*, pages 293–307, 2015. 5.6.2, 5.8.2

[201] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proceedings of the 17th IEEE International Conference on Data Engineering, ICDE '01*, pages 567–574, 2001. 2.3

[202] S. Pantela and S. Idreos. One Loop Does Not Fit All. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, pages 2073–2074, 2015. 8.2

[203] Y. Papakonstantinou, V. R. Borkar, M. Orgiyan, K. Stathatos, L. Suta, V. Vassalos, and P. Velikhov. XML queries and algebra in the Enosys integration platform. *Data & Knowledge Engineering*, 44(3):299–322, 2003. 2.2, 4.3

[204] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. Sql-and operator-centric data analytics in relational main-memory databases. *Proceedings of the 20th International Conference on Extending Database Technology, EDBT '17*, pages 84–95, 2017. 2.5.2

[205] J. Paul, J. He, and B. He. GPL: A GPU-based Pipelined Query Processing Engine. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD '16*, pages 1935–1950, 2016. 7.4

[206] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten. CPU and Cache Efficient Management of Memory-Resident Databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE '13*, pages 14–25, 2013. 2.3

[207] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. *Proceedings of the VLDB Endowment*, 5(11):1447–1458, 2012. 7.2

[208] PostgreSQL. JSON Types. `http://www.postgresql.org/docs/9.5/static/datatype-json.html`. A.1

[209] N. Prokoshyna, J. Szlichta, F. Chiang, R. J. Miller, and D. Srivastava. Combining quantitative and logical data cleaning. *Proceedings of the VLDB Endowment*, 9(4):300–311, 2015. 2.5.2

[210] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K. Sattler. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In *6th TPC*

Technology Conference on Performance Characterization and Benchmarking. Traditional to Big Data, TPCTC '14, 2014. 1.2, 7.1, 7.2, 7.5.1

[211] L. Qiao, Y. Li, S. Takiar, Z. Liu, N. Veeramreddy, M. Tu, Y. Dai, I. Buenrostro, K. Surlaker, S. Das, and C. Botev. Gobblin: Unifying Data Ingestion for Hadoop. *Proceedings of the VLDB Endowment*, 8(12):1764–1775, 2015. 5.2

[212] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *Proceedings of the VLDB Endowment*, pages 381–390, 2001. 2.5, 2.5.2, 6.1

[213] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *Proceedings of the 22nd IEEE International Conference on Data Engineering, ICDE '06*, page 23, 2006. 2.3, 4.2, 7.3

[214] Robert Kruszewski. *Catalyst: Allow adding custom optimizers*, 2016. 3

[215] K. A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '02*, pages 109–120, 2002. 8.2

[216] M. T. Roth, F. Ozcan, and L. M. Haas. Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *Proceedings of the 25th International Conference on Very Large Data Bases, VLDB '99*, pages 599–610, 1999. 4.5.2, 5.2

[217] M. T. Roth and P. M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97*, pages 266–275, 1997. 4.2, 4.5.2, 5.2, 5.6.1

[218] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00*, pages 249–260, 2000. 4.6

[219] F. Rusu. Scalable in-situ exploration over raw data. In *8th Biennial Conference on Innovative Data Systems Research, CIDR '17*, 2017. 2.1.3

[220] A. D. Sarma, Y. He, and S. Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *Proceedings of the VLDB Endowment*, 7(12):1059–1070, 2014. 6.3.1, 6.3.2

[221] K.-U. Sattler, S. Conrad, and G. Saake. Adding conflict resolution features to a query language for database federations. *Australasian Journal of Information Systems, AJIS '00*, 8(1), 2000. 2.5.2

[222] F. M. Schuhknecht, J. Dittrich, and A. Sharma. RUMA Has It: Rewired User-space Memory Access is Possible! *Proceedings of the VLDB Endowment*, 9(10):768–779, 2016. 7.5.1

[223] P. M. Schwarz, W. Chang, J. C. Freytag, G. M. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst Database System. In *1986 International Workshop on Object-Oriented Database Systems, OODBS '86*, pages 85–92, 1986. 2.1.3

[224] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to Architect a Query Compiler. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD '16*, pages 1907–1922, 2016. 2.3

[225] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases, VLDB '99*, 1999. 4.1, 4.2

[226] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST '12*, pages 1–10, 2010. 2.1.3

[227] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B. P. Hsu, and K. Wang. An Overview of Microsoft Academic Service (MAS) and Applications. Proceedings of the 24th International Conference on World Wide Web Companion, WWW '15, pages 243–246, 2015. 6.7

[228] P. Sioulas and A. Ailamaki. Vectorizing an In Situ Query Engine. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD '16*, pages 2261–2262, 2016. 8.2

[229] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, SIGMOD '85*, pages 236–246, 1985. 1

[230] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A Modular Query Optimizer Architecture for Big Data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 337–348, 2014. 5.8.2

[231] J. Sompolski. Just-in-time Compilation in Vectorized Query Execution. Master's thesis, University of Warsaw, VU University Amsterdam, 2011. 8.2

[232] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN '11*, pages 33–40, 2011. 8.2

[233] M. Stonebraker. Technical perspective - One size fits all: an idea whose time has come and gone. *Communications of ACM*, 51(12):76, 2008. 1.2, 3.3, 4.1

[234] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A

Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05*, pages 553–564, 2005. 3.3

[235] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data Curation at Scale: The Data Tamer System. In *6th Biennial Conference on Innovative Data Systems Research, CIDR '13*, 2013. 2.5, 2.5.2, 6.1

[236] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: a SQL system for multi-structured data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 815–826, 2014. 4.2

[237] The HDF Group. HDF5. `http://www.hdfgroup.org/HDF5`. 3.4.1

[238] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009. 4.1, 5.1, 5.2, 5.3, 5.3

[239] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering, TKDE '98*, 10(5):808–823, 1998. 4.2, 5.2

[240] P. W. Trinder. Comprehensions, a Query Notation for DBPLs. In *3rd International Workshop on Database Programming Languages: Bulk Types and Persistent Data*, 1991. 4.3

[241] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13*, 2013. 7.5

[242] Unidata. NetCDF. `http://www.unidata.ucar.edu/software/netcdf/`. 2.1.3

[243] P. Vassiliadis. A Survey of Extract-Transform-Load Technology. In *Integrations of Data Warehousing, Data Mining and Database Technologies - Innovative Approaches.*, pages 171–199. 2011. 5.2

[244] P. Vassiliadis and A. Simitsis. Near Real Time ETL. In *New Trends in Data Warehousing and Data Analysis*, pages 1–31. 2009. 5.2

[245] T. Venetis, A. Ailamaki, T. Heinis, M. Karpathiotakis, F. Kherif, A. Mitelpunkt, and V. Vassalos. Towards the Identification of Disease Signatures. In *8th International Conference on Brain Informatics and Health, BIH '15*, pages 145–155, 2015. 1

[246] R. Verborgh and M. D. Wilde. *Using OpenRefine*. Packt Publishing, 2013. 2.5, 2.5.2

[247] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software, TOMS '85*, 11(1):37–57, 1985. 6.3.2

[248] D. Šidlauskas, C. S. Jensen, and S. Šaltenis. A Comparison of the Use of Virtual Versus Physical Snapshots for Supporting Update-intensive Workloads. In *Proceedings of the 8th International Workshop on Data Management on New Hardware, DaMoN '12*, pages 1–8, 2012. 7.3, 7.5.1, 7.5.1

[249] P. Wadler. Comprehending Monads. volume 2, pages 461–493, 1992. 2.2, 2.2

[250] S. Wanderman-Milne and N. Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin*, 37(1):31–37, 2014. 2.3

[251] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria Big Data Management and Analytics System and Cloud Services. In *8th Biennial Conference on Innovative Data Systems Research, CIDR '17*, 2017. 5.2

[252] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A Sample-and-clean Framework for Fast and Accurate Query Processing on Dirty Data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 469–480, 2014. 2.5.2, 1

[253] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *Operating Systems Review*, 43(2):76–85, 2009. 2.4.2, 7.1

[254] L. Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000. 2.2

[255] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. G. R. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Laurent, J. Meredith, P. Messmer, E. Otoo, V. Perevoztchikov, A. Poskanzer, Prabhat, O. Rübel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W.-M. Zhang. FastBit: interactively searching massive data. *Journal of Physics Conference Series, Proceedings of SciDAC*, 180(1):012053, 2009. 2.1.3

[256] R. Xin. Made sort-based shuffle the default implementation, Spark Issue 3280, 2014. 6.7.3

[257] Y. Xu, Y. Du, Y. Zhang, and J. Yang. A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs. In *Proceedings of the 25th International Conference on Supercomputing, ICS '11*, pages 285–294, 2011. 2.4.2

[258] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013. 2.4.1, 7.4

[259] N. Yuhanna. The Forrester Wave™: Enterprise Data Virtualization, Q1 2015. 1.3.1

[260] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI '12*, 2012. 2.5.2, 5.1, 5.3, 6.2.2, 6.5, 6.6

Manolis (Manos) Karpathiotakis

Ph.D. in Computer Science
École Polytechnique Fédérale de Lausanne
CH-1015, Lausanne, Switzerland
manos.karpathiotakis@epfl.ch
http://karpathiotakis.net

RESEARCH INTERESTS

data management in the presence of data variety, query processing, just-in-time code generation, (scale-out) data analytics, hardware accelerators

ACADEMIC BACKGROUND

Ph.D. in Computer Science, 2012 – 2017 **École Polytechnique Fédérale de Lausanne**
Thesis: Just-in-time Analytics Over Heterogeneous Data and Hardware
Advisor: Prof. Anastasia Ailamaki

M.Sc. in Advanced Information Systems, 2008 – 2011 **University of Athens, Greece**
Grade: 9.34 / 10
Thesis: Design and Implementation of a Registry for the Semantic Sensor Web

B.Sc. in Informatics & Telecommunications, 2004 – 2008 **University of Athens, Greece**
Grade: 8.35 / 10 - Graduation Rank: 3rd / 52
Thesis: e-Government with the utilization of GIS technologies

HONORS & AWARDS

- EPFL Teaching Assistant Award, 2016
- IBM PhD Fellowship Award, 2015-2016
- EPFL Computer Science Fellowship, 2012-2013
- Awarded 3rd place in Semantic Web Challenge 2012

WORKING EXPERIENCE

09/2012 – present **École Polytechnique Fédérale de Lausanne** **Lausanne, Switzerland**
- Doctoral Assistant & Member of the Data-Intensive Applications and Systems (DIAS) laboratory, advised by Prof. Anastasia Ailamaki.
- Working on analytics over heterogeneous data and hardware (http://dias.epfl.ch/vida).

06/2015 – 09/2015 **IBM Almaden Research Center** **San Jose, California**
- Intern at the Big Data Research Group, mentored by Avrilia Floratou and Fatma Özcan.
- Worked on enhancing the federated querying capabilities of Spark, a big data engine.

10/2013 – 06/2015 **École Polytechnique Fédérale de Lausanne** **Lausanne, Switzerland**
- Research Assistant in the **Human Brain Project** (https://www.humanbrainproject.eu).
- Worked on the query engine of a platform used for the diagnosis of brain diseases.

06/2009 – 09/2012 **University of Athens** **Athens, Greece**
- Implementation team coordinator, Research Assistant, Scientific Programmer in the context of the European FP7 Programs **SemsorGrid4Env** (http://semsorgrid4env.eu) and **TELEIOS** (http://www.earthobservatory.eu).
- Worked on a scalable spatiotemporal RDF store (http://www.strabon.di.uoa.gr).

PUBLICATIONS

Conferences:

- M. Karpathiotakis, A. Floratou, F. Ozcan, A. Ailamaki, **"No Data Left Behind: Real-Time Insights from a Complex Data Ecosystem"**. In Proceedings of the ACM Symposium on Cloud Computing ***(SoCC)***, 2017
- S. Giannakopoulou, M. Karpathiotakis, B. Gaidioz, A. Ailamaki, **"CleanM: An Optimizable Query Language for Unified Scale-Out Data Cleaning".** In Proceedings of the Very Large Databases Endowment (***PVLDB)***, Vol.10(11), 2017
- M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, A. Ailamaki, **"Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing".** In Proceedings of the Very Large Databases Endowment (***PVLDB)***, Vol.10(10), 2017
- R. Appuswamy, M. Karpathiotakis, D. Porobic, A. Ailamaki, **"The Case For Heterogeneous HTAP"**. In Conference on Innovative Data Systems Research (***CIDR)***, 2017
- M. Karpathiotakis, I. Alagiannis, A. Ailamaki, **"Fast Queries Over Heterogeneous Data Through Engine Customization"**. In Proceedings of the Very Large Databases Endowment (***PVLDB)***, Vol.9(12), 2016
- T. Venetis, A. Ailamaki, T. Heinis, M. Karpathiotakis, F. Kherif, A. Mitelpunkt, V. Vassalos, **"Towards the Identification of Disease Signatures"**. In International Conference on Brain Informatics and Health (***BIH)***, 2015
- M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, A. Ailamaki, **"Just-In-Time Data Virtualization: Lightweight Data Management with ViDa"**. In Conference on Innovative Data Systems Research (***CIDR)***, 2015
- M. Karpathiotakis, M. Branco, I. Alagiannis, A. Ailamaki, **"Adaptive Query Processing on RAW Data"**. In Proceedings of the Very Large Databases Endowment ***PVLDB***, Vol.7(12), 2014
- K. Kyzirakos, M. Karpathiotakis, K. Bereta, G. Garbis, C. Nikolaou, P. Smeros, S. Giannakopoulou, K. Dogani, M. Koubarakis, **"The Spatiotemporal RDF Store Strabon"**. In International Symposium of Advances in Spatial and Temporal Databases (***SSTD)***, 2013
- K. Kyzirakos, M. Karpathiotakis, M. Koubarakis, **"Strabon, a Semantic Geospatial DBMS"**. In International Semantic Web Conference (***ISWC)***, 2012
- M. Koubarakis, M. Sioutis, K. Kyzirakos, M. Karpathiotakis, C. Nikolaou, S. Vassos, G. Garbis, K. Bereta, O. C. Dumitru, D. E. Molina, K. Molch, G. Schwarz, M. Datcu, **"Building Virtual Earth Observatories using Ontologies, Linked Geospatial Data and Knowledge Discovery Algorithms"**, In International Conference on Ontologies, DataBases, and Applications of Semantic (***ODBASE)***, 2012
- A. J. G. Gray, R. Garcia-Castro, K. Kyzirakos, M. Karpathiotakis, J.-P. Calbimonte, K. Page, J. Sadler, A. Frazer, I. Galpin, A. Fernandes, N. W. Paton, O. Corcho, M. Koubarakis, D. De Roure, K. Martinez, A. Gomez-Perez, **"A Semantically Enabled Service Architecture for Mashups over Streaming and Stored Data"**. In Extended Semantic Web Conference (***ESWC)***, 2011

Refereed Journals:

- C. Nikolaou, K. Dogani, K. Bereta, G. Garbis, M. Karpathiotakis, K. Kyzirakos, M. Koubarakis, **"Sextant: Visualizing time-evolving liked geospatial data"**. In ***Journal of Web Semantics*** 35
- K. Kyzirakos, M. Karpathiotakis, G. Garbis, C. Nikolaou, K. Bereta, I. Papoutsis, T. Herekakis, D. Michail, M. Koubarakis, C. Kontoes, **"Wildfire monitoring using satellite images, ontologies and linked geospatial data"**. In ***Journal of Web Semantics*** 24
- J. G. Gray, J. Sadler, O. Kit, K. Kyzirakos, M. Karpathiotakis, J.-P. Calbimonte, K. Page, R. García-Castro, A. Frazer, I. Galpin, A. Fernandes, N.W. Paton, O. Corcho, M. Koubarakis, D. De Roure, K. Martinez, A. Gómez-Pérez. **"A Semantic Sensor Web for Environmental Decision Support Applications"**. In ***Sensors*** 11(9), 2011

Demonstrations:

- K. Bereta, C. Nikolaou, M. Karpathiotakis, K. Kyzirakos, M. Koubarakis, **"SexTant: Visualizing Time-Evolving Linked Geospatial Data"**. In International Semantic Web Conference (***ISWC)***, 2013
- K. Kyzirakos, M. Karpathiotakis, G. Garbis, C. Nikolaou, K. Bereta, M. Sioutis, I. Papoutsis, T. Herekakis, D. Michail, M. Koubarakis, C. Kontoes, **"Real Time Fire Monitoring Using Semantic Web and Linked Data Technologies"**. In International Semantic Web Conference (***ISWC)***, 2012
- M. Koubarakis, K. Kyzirakos, M. Karpathiotakis, C. Nikolaou, S. Vassos, G. Garbis, M. Sioutis, K. Bereta, D. Michail, C. Kontoes, I. Papoutsis, T. Herekakis, S. Manegold, M. Kersten, M. Ivanova, H. Pirk, Y. Zhang, M. Datcu, G. Schwarz, O. C. Dumitru, D. Espinoza Molina, K. Molch, U. Di Giammatteo, M. Sagona, S. Perelli, T. Reitz, E. Klien, R. Gregor, **"TELEIOS: A Database-Powered Virtual Earth Observatory"**. In Proceedings of the Very Large Databases Endowment (***PVLDB)***, Vol.5(12), 2012

Workshops:

- A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, A. Ailamaki, **"DBMS Data Loading: An Analysis on Modern Hardware"**. In International Workshop on Accelerating Data Analysis and Data Management Systems Using Modern Processor and Storage Architectures (***ADMS)***, 2016
- C. Kontoes, I. Keramitsoglou, I. Papoutsis, D. Michail, T. Herekakis, P. Xofis, M. Koubarakis, K. Kyzirakos, M. Karpathiotakis, C. Nikolaou, M. Sioutis, G Garbis, S. Vassos, S. Manegold, M. Kersten, H. Pirk, M. Ivanova, **"Operational Wildfire Monitoring and Disaster Management Support Using State-of-the-art EO and Information Technologies"**, In International Workshop on Earth Observation and Remote Sensing Applications (***EORSA***), 2012
- M. Koubarakis, K. Kyzirakos, M. Karpathiotakis, C. Nikolaou, M. Sioutis, S. Vassos, D. Michail, T. Herekakis, C. Kontoes, I. Papoutsis, **"Challenges for Qualitative Spatial Reasoning in Linked Geospatial Data"**. In Benchmarks and Applications of Spatial Reasoning ***(BASR)***, 2011
- K. Kyzirakos, M. Karpathiotakis, M. Koubarakis, **"Developing Registries for the Semantic Sensor Web using stRDF and stSPARQL"**. In International Workshop on Semantic Sensor Networks (***SSN)***, 2010

Tutorials:

- M. Koubarakis, K. Kyzirakos, M. Karpathiotakis, **"Data models, Query Languages, Implemented Systems and Applications of Linked Geospatial Data"**. In Extended Semantic Web Conference ***(ESWC)***, 2012
- M. Koubarakis, M. Karpathiotakis, K. Kyzirakos, C. Nikolaou, M. Sioutis, **"Data Models and Query Languages for Linked Geospatial Data"**. In Reasoning Web (***RW)***, *2012*

Contests:

- K. Kyzirakos, M. Karpathiotakis, G. Garbis, C. Nikolaou, K. Bereta, I. Papoutsis, T. Herekakis, D. Michail, M. Koubarakis, C.Kontoes: **"Wildfire Monitoring Using Satellite Images, Ontologies, and Linked Geospatial Data"**. In **Semantic Web Challenge 2012**, in conjunction with International Semantic Web Conference (**ISWC**) 2012 (**Awarded 3rd place**)

CONFERENCE PRESENTATIONS & INVITED TALKS

- "Data models, Query Languages, Implemented Systems and Applications of Linked Geospatial Data". At **ESWC** 2012
- "Adaptive Query Processing on RAW Data". At **VLDB** 2014
- "Just-In-Time Data Virtualization: Lightweight Data Management with ViDa". At **CIDR** 2015
- "Just-In-Time Data Virtualization: Lightweight Data Management with ViDa". At **IBM**, 2015
- "Just-In-Time Data Virtualization". At **Ecocloud Annual Event** 2016
- "Fast Queries Over Heterogeneous Data Through Engine Customization". At **VLDB** 2016
- "DBMS Data Loading: An Analysis on Modern Hardware". At **ADMS** 2016
- "Just-in-time Database Engines". At **IBM**, 2017
- "Just-in-time Database Engines". At **Microsoft**, 2017
- "Just-in-time Analytics in the presence of Heterogeneity". At **Huawei**, 2017

PROFESSIONAL ACTIVITIES

Reviewer: SIGMOD 2014 (External), Semantic Web Journal

PROFESSIONAL MEMBERSHIPS

- ACM
- IEEE

IN THE NEWS

- https://actu.epfl.ch/news/manos-karpathiotakis-wins-ibm-fellowship-award-2/
- http://www.ecocloud.ch/2015/03/12/manos-karpathiotakis-wins-the-ibm-fellowship-award/

TEACHING ASSISTANTSHIPS

Spring 2014-2017	*Introduction to Database Systems*
Fall 2016	*Computer Architecture*
Fall 2013	*Functional Programming Principles in Scala*
Fall 2013	*Principles of Reactive Programming*
Fall 2009-2011	*Knowledge Technologies*
Spring 2010	*Foundation of* Databases
Fall 2008,2009	*Introduction to Programming*
Fall 2008	*Operating Systems*

LANGUAGES

Greek (Native), English (Fluent), French (Beginner)

REFERENCES

Provided upon request.