Declarative Nested Data Transformations at Scale and Biomedical Applications

Jaclyn Marjorie Smith

Wolfson College

University of Oxford

A thesis presented for the degree of

Doctor of Philosophy in Computer Science

Michaelmas 2021

This thesis is dedicated to my family and friends,

Acknowledgments

I would like to thank my supervisor Michael Benedikt for always being genuinely involved and facilitating a collaborative and encouraging environment. Thank you to Milos Nikolic for helping me with the mysteries of Spark and fueling my interest in systems development. I express my deepest appreciation to you both. Together you have helped shape a project that allowed me to grow as a database researcher, solidify my coding skills, and fuel my interest in biology. Thank you also to Amir Shaikhha for the extreme programming sessions and the crash courses in Scala. To Boris Motik and Pramod Bhatotia, thank you for the invaluable feedback on this thesis. Many thanks to Brandon Moore, Yao Shi, and Pavlos Piperis for all your hard work. I very much appreciate the time and inputs I have received from all the examiners along the way. To all those mentioned and others at Oxford, thank you for showing me a wonderful side to research.

None of this would have been possible without the love and support of my family. Thank you to my mom, who never ceases to amaze me - whose strength and hard work have forever shaped me. To my dad, who will always be with me in spirit. To Em, for always providing your undeniable support - I am forever grateful. Special thanks to everyone at ODA for always being an inspiration. To all my friends and loved ones who have shown me the beauty in discovering more about the world, I am truly grateful for you all.

Abstract

While large-scale distributed data processing platforms have become an attractive target for query processing, these systems are problematic for applications that deal with nested collections. Programmers are forced either to perform non-trivial translations of collection programs or to employ automated flattening procedures, both of which lead to performance problems. These challenges only worsen for nested collections with skewed cardinalities, where both handcrafted rewriting and automated flattening are unable to enforce load balancing across partitions.

In this work, the TraNCE compilation framework is proposed that translates a program manipulating nested collections into a set of semantically equivalent shredded queries that can be efficiently evaluated. The framework employs a combination of query compilation techniques, an efficient data representation for nested collections, and automated skew-handling. Biomedical case studies are presented that outline research and clinical applications for the platform, including data integration support for building feature sets for classification. An extensive experimental evaluation is provided using both synthetic and real-world dataset from the biomedical domain. The evaluation shows that the system is capable of outperforming the common alternative, based on "flattening" complex data structures, and runs efficiently when alternative approaches are unable to perform at all.

Contents

1 Introduction **1**

1.1 Challenge 1: Programming mismatch 4

1.2 Challenge 2: Partitioning Nested Data 6

1.3 Challenge 3: Data Skew 7

1.4 Existing Solutions 7

1.5 Proposed Solutions 8

1.5.1 Compilation of NRC Queries with Aggregation 9

1.5.2 Query and Data Shredding 9

1.5.3 Automated Skew-Handling 10

1.5.4 Additional Contributions 11

1.6 Thesis-Related Collaborations and Publications 11

1.7 Thesis Organization 11

2 Foundations of the Framework and Background **13**

2.1 Nested Data Model 14

2.2 Nested Relational Calculus 15

2.2.1 A Brief History 17
2.2.2 Comprehension-Style Syntax 17
2.3 Plan Language 18
2.4 Plan Translation 19
2.5 Query and Data Shredding 24
2.5.1 Shredded Representation 24
2.5.2 Data Shredding and Unshredding 26
2.5.3 Query Shredding 28
2.6 Distributed Processing Frameworks 33
2.7 Skew-Handling Techniques 36
2.8 Implementation of Nested Query languages 38
2.8.1 Query Language Implementations 38
2.8.2 Internal Data-Specific Systems 39
2.8.3 Distributed Processing-Specific Systems 39
2.9 User-Defined Functions 40
2.10 Thesis Motivation 41

3 Compiling NRC with Aggregation **42**

3.1 NRC with Aggregation 42
3.1.1 Semantics 44
3.2 Plan Language Extensions 45
3.3 Plan Translation Extensions 45

3.4 Code Generation 51

4 Query and Data Shredding **55**

4.1 Shredded NRC 57

4.2 Query Shredding Algorithm 58

4.2.1 Symbolic query shredding 59

4.2.2 Materialization 61

4.2.3 Discussion of Sequential Shredding 65

4.3 Domain Elimination 66

4.4 Relational View 67

4.5 Plan Translation 69

4.6 Code generation 70

5 Skew-Handling and Additional Optimizations **72**

5.1 Skew-Resilient Processing 72

5.1.1 Heavy Key Identification 73

5.1.2 Skew-Aware Plans 74

5.2 Plan Optimizations 75

5.3 Code Generation Optimizations 79

5.3.1 Partial Aggregation and Unnest Merge 79

5.3.2 Nest and Join Merge 80

6 TraNCE Architecture and Additional Features **82**

6.1 Input Definitions and Sanitation 83
6.2 Interactive Frontend 84
6.3 User-Defined Functions 87
6.3.1 Compilation Extensions 87
6.3.2 External Types 88
6.3.3 Shredding UDFs 89
6.3.4 UDF Hints 91

7 Performance Comparison of Nested Data Systems 93

7.1 Our Nested TPC-H Benchmark 95
7.1.1 Flat-to-nested 96
7.1.2 Nested-to-nested 98
7.1.3 Nested-to-flat 98
7.2 Experimental Setup 98
7.2.1 Systems Evaluated 101
7.2.2 Additional Systems Explored 103
7.2.3 Experimental strategies 105
7.3 Experimental Results for Non-Skewed Inputs 105
7.3.1 Flat-to-nested 109
7.3.2 Nested-to-nested 111
7.3.3 Nested-to-flat 113
7.4 Experimental Results for Skewed Inputs 114

7.4.1 Skew-handling Overhead 117

7.4.2 Increasing Amounts of Skew 117

7.4.3 Skew-handling for the Shredded Representation 118

7.5 Experiments Summary 119

8 Biomedical Analysis and Performance in TraNCE 122

8.1 Background 123

8.2 Biomedical Data Sources 125

8.3 Clinical Exploratory Queries 129

8.3.1 Runtime performance 131

8.4 Multi-Omics Driver Gene Identification 132

8.4.1 Runtime performance 137

8.5 Mutational Burden Predictions 138

8.5.1 Classification with burden-based features 140

8.6 Additional TraNCE Performance Results 145

8.6.1 Scalability 145

8.6.2 Skew-handling 148

8.7 Sharing in the shredded representation 150

8.8 Future Aims for Biomedical Analysis Support 152

9 Feature Selection Filters: A Biomedical Case Study 153

9.1 Background 155

9.1.1 Chi-square 155

9.1.2 ANOVA 156

9.1.3 Mutual Information 156

9.1.4 Recursive Feature Elimination 157

9.1.5 MultiSURF 157

9.1.6 Feature Selection in the Literature 158

9.2 Partial Filter Calculations 158

9.2.1 Correlation Filter 159

9.2.2 Chi-square Filter 160

9.3 Implementation 161

9.3.1 Hints 164

9.4 Use Cases 165

9.4.1 Binary Classification: Prostate Severity 165

9.4.2 Multi-class Prediction: Tumor Site of Origin 168

9.5 Experiments 168

9.5.1 Experimental Setup 168

9.5.2 Binary Classification: Prostate Severity 170

9.5.3 Multi-class Prediction: Tumor Site of Origin 177

9.5.4 Gene Enrichment Analysis 183

9.6 Future Exploration 187

10 Conclusion **189**

A Appendices **196**

A.1 Spark RDDs vs Spark Datasets 196

A.2 Standard compilation framework optimizations 198

A.3 One-vs-rest binary models for gene burdens based on raw counts 200

A.4 Feature selection experiments: preliminary exploration 204

List of Figures

1.1 Example data distribution for `COP` on a distributed processing platform. . 2

1.2 Flattened `COP` resulting from example Spark program. 6

2.1 Plan translation rules translating NRC^{+} to plan language operators. . . . 21

2.2 Syntax of $\mathrm{NRC}^{Lbl+\lambda}$. 29

2.3 Semantics of $\mathrm{NRC}^{Lbl+\lambda}$. 29

2.4 Query shredding algorithm for NRC 30

2.5 Example SparkSQL application that uses the Dataset API to join flattened `COP` with `Part` and then group by customer name and order date. . 34

2.6 Equivalent SparkSQL program from Figure 2.5 defined using SQL instead of the Dataset API. 35

3.1 Syntax of NRC_{agg}. 43

3.2 Semantics of the NRC_{agg} extensions. 44

3.3 Aggregate plan language operators and their semantics. 45

3.4 Plan translation rules for the aggregation operations `sumBy`, `groupBy`, and `dedup`. 46

3.5 A query plan for the `Totals` example with output types. Bag types are annotated with ^{B}. 51

3.6 Plan language operators and their semantics for Spark Datasets. 53

3.7 Plan language operators and their semantics for Spark RDDs. 54

4.1 The shredded data of `COP` encoded as label/bag pairs (recursive view) with one top-level $\texttt{COP}^{\mathrm{F}}$ and two value dictionaries $\texttt{COP}^{\mathrm{F}}_{\texttt{corders}}$ and $\texttt{COP}^{\mathrm{F}}_{\texttt{corders_oparts}}$ 56

4.2 The syntax of $\mathrm{NRC}^{Lbl+\lambda}_{agg}$. 57

4.3 Semantics of $\mathrm{NRC}^{Lbl+\lambda}_{agg}$, following Figure 2.3. 57

4.4 Query shredding algorithm following from Figure 2.4 with aggregation extensions. 59

4.5 Materialization algorithms. 62

4.6 The shredded data of `COP` in relational view with one top-level $\texttt{COP}^{\mathrm{F}}$ and two value dictionaries $\texttt{COP}^{\mathrm{F}}_{\texttt{corders}}$ and $\texttt{COP}^{\mathrm{F}}_{\texttt{corders_oparts}}$ 68

4.7 The shredded `Totals` query in relational view with one top-level $\texttt{COP}^{\mathrm{F}}$ and two value dictionaries $\texttt{COP}^{\mathrm{F}}_{\texttt{corders}}$ and $\texttt{COP}^{\mathrm{F}}_{\texttt{corders_oparts}}$ 68

4.8 The generated Spark program for the shredded `Totals` query following relational view. The plans resulting from the plan translation procedure for both dictionaries are also provided. 71

5.1 Skew-aware implementation for the plan language operators using Spark Datasets. 76

5.2 Local aggregation optimization 77

5.3 The optimized query plan for the `Totals`running example. Index operator removal, selections pushed, and local aggregation optimizations are colored red. 78

6.1 System architecture. 84

6.2 Query builder view 85

6.3 Compilation view 85

6.4 Results view 86

7.1 Flat-to-nested query with four levels of nesting, from which queries with zero to three levels of nesting can be derived. Ellipses represent the additional fields that may be present based on narrow and wide schemas. . . 97

7.2 Nested-to-nested query with four levels of nesting. 99

7.3 Nested-to-flat query with four levels of nesting. 100

7.4 Performance comparison of flat-to-nested queries including all competitors. MongoDB and Citus failed for all wide variants so are not included. 106

7.5 Performance comparison of narrow nested-to-nested and nested-to-flat queries for all competitors. 107

7.6 Performance comparison of the narrow and wide benchmarked TPC-H queries with varying levels of nesting (0-4). Each run is marked with total shuffle memory (GB). 108

7.7 Performance comparison with and without skew-resilience for increasing amounts of skew. 115

7.8 Heavy key identification overhead for the sampling method. 116

7.9 Amount of shuffle related to the join on skewed data. 116

8.1 The NRC_{agg} programs for clinical exploration. Ellipses represent additional attributes from `Occurrences` and `CopyNumber`. 130

8.2 Results for the clinical exploration programs. The standard compilation route fails for all runs with the Pancancer dataset. 133

8.3 Summary of the cancer driver gene analysis. The pipeline starts by integrating somatic mutations and copy number variation, further integrates network information, and gene expression data. The genes with the highest connectivity scores are taken to be drivers. 133

8.4 The NRC_{agg} program of the driver gene analysis. 135

8.5 Runtimes for each of the stages of the driver gene analysis. The standard compilation fails at the `SampleNetwork` program. 137

8.6 Workflow diagram representing the burden-based analyses for both genes and pathways, and downstream classification problem. The results of the pathway burden analysis feed into a classification analysis using multi-class and one-vs-rest methods to predict tumor of origin. . . . 139

8.7 The burden-based programs for feature vector construction. 140

8.8 Accuracy and loss of the multi-class neural network for tumor tissue site. 141

8.9 Accuracy and loss for the tumor tissue site based binary network, includes results for the three worst-performing classes from the multi-class network. 142

8.10 Runtime burden-based analyses for an increasing number of top-level variants. 146

8.11 Scalability for the driver gene analysis measured using `HybridMatrix` and `SampleNetwork` programs produced from the shredded compilation route for a variety of cluster configurations. 147

8.12 Performance comparison of the skew-handling techniques for both the standard and shredded compilation routes. Queries are organized based on increasing amounts of skew, such that tumor sites is representative of low skew and gene families of high skew. 151

9.1 Example shredded NRC_{agg} program and shredded function definition for `trainudf`. 163

9.2 The NRC_{agg} programs for the feature selection case study. 166

9.3 Runtimes of the feature vector construction, feature selection methods, and applied filter for `GeneImpactBurden`, `GeneExprBurden`, and `GeneMultiBurden` programs. 172

9.4 Runtimes of the feature vector construction, feature selection methods, and applied filter for `GeneBurdenPancan` program. 178

9.5 Confusion matrix of the RFE model. The cancer types are encoded as [Colon = 0, Breast = 1, Lung = 2, Kidney = 3, Stomach = 4, Ovary = 5, Endometrial = 6, Head and Neck = 7, Central nervous system = 8] . . 181

9.6 Gene set enrichment analysis for overlapping RFE and ANOVA feature sets. High-confidence associations are in dark blue. Generated by WebGestalt. 184

9.7 Biological processes, cellular components, and molecular function categories for the overlapping gene set of ANOVA/RFE. Generated by WebGestalt. 185

9.8 Volcano plot for the 114 overlapping ANOVA genes used in the one-vs-rest experiment. High-confidence associations are labeled by definition. Generated by WebGestalt. 186

9.9 Represented biological processes, cellular components, and molecular function categories for the 114 overlapping ANOVA genes used in one-vs-rest. Generated by WebGestalt. 186

10.1 Partial shredding for the `COP` input and the `Totals`running example. . . . 191

A.1 Performance comparison of RDDs verse Datasets for the nested-to-nested benchmarked queries. 197

A.2 Performance comparison of benchmarked queries for increasing optimization levels of the standard compilation route. 199

A.3 The accuracy and loss of the binary neural network for breast. 201

A.4 The accuracy and loss of the binary neural network for colon. 201

A.5 The accuracy and loss of the binary neural network for endometrial. . . . 202

A.6 The accuracy and loss of the binary neural network for kidney. 202

A.7 The accuracy and loss of the binary neural network for lung. 203

A.8 The accuracy and loss of the binary neural network for ovary. 203

A.9 Overfitting behavior in the training and validation error of the model using `GeneImpactBurden` and chi-square feature selection returning top 20000 204

A.10 Overfitting behavior in the training and validation error of the model using `GeneImpactBurden` and ANOVA feature selection returning top 20000. 205

Chapter 1

Introduction

Large-scale, distributed data processing platforms such as Spark [1], Flink [2], and Hadoop [3] have become indispensable tools for modern data analysis. These frameworks work on top of a cluster of machines where one is designated as the central, or *coordinator* node, and the other nodes are *workers*. An application is submitted to the coordinator node, which then delegates tasks to worker nodes in a highly distributed, parallel fashion. The wide adoption of these platforms stems from powerful functional-style APIs that allow programmers to express complex analytical tasks while abstracting distributed resources and data parallelism.

Distributed processing frameworks use an underlying *complex data* model that allows for data to be described as a collection of tuples whose values may themselves be collections. This *nested data* representation arises naturally in many domains, such as web, biomedicine, and business intelligence. The widespread use of nested data also contributed to the rise in NoSQL databases [4, 5, 6], where the nested data model is central. Thus, it comes as no surprise that nested data accounts for most large-scale, structured data processing at major web companies [7].

Despite natively supporting nested data, existing distributed systems fail to harness the full potential of processing nested collections at scale. One implementation difficulty is the discrepancy between the tuple-at-a-time processing of local programs and the bulk processing used in distributed settings. Though maintaining similar APIs, programs that use local collections often require non-trivial and error-prone trans-

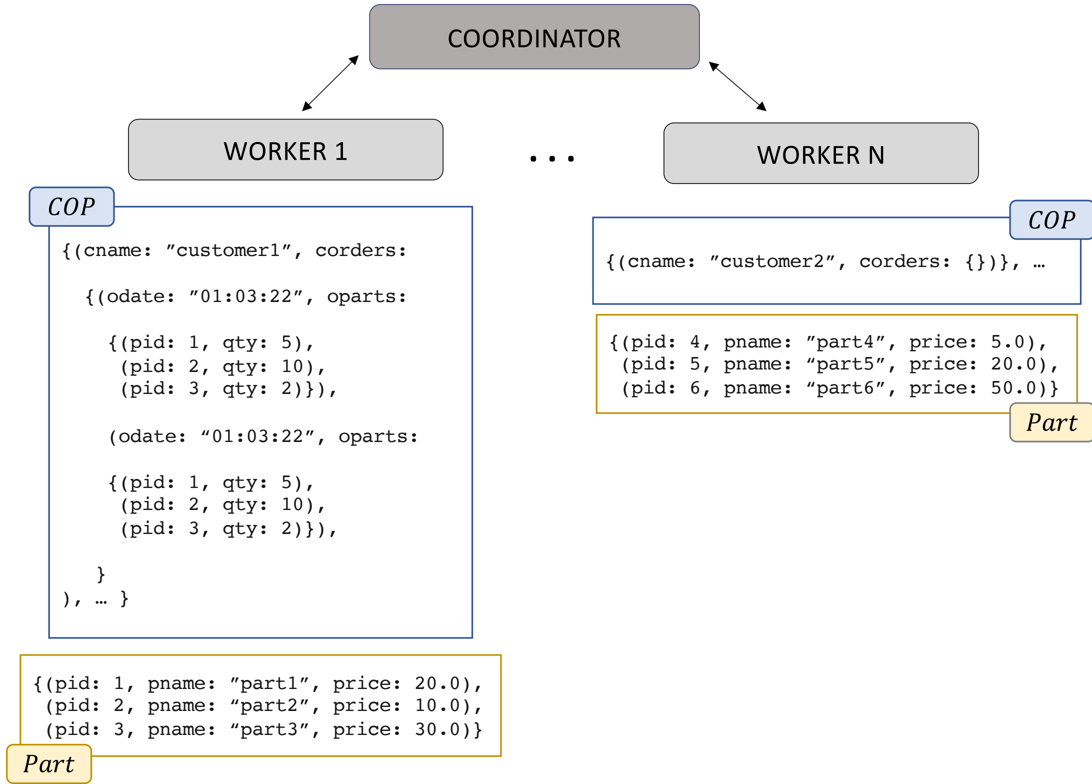


Figure 1.1: Example data distribution for COP on a distributed processing platform.

lations to distributed collection programs. Beyond programming difficulties, these frameworks use a *top-level distribution strategy* that treats tuples as objects, persisting nested inner collections in the same location as the top-level information. This strategy hinders the distribution of nested data and prevents distributed frameworks to scale for large nested data and skewed inner collections. These difficulties are elaborated next by means of example.

Consider a variant of the TPC-H database containing a flat relation `Part` with information about parts and a nested collection `COP` with information about customers, their orders, and the parts purchased in the orders. The `COP` dataset stores customer names, order dates, part IDs and purchased quantities per order. The `COP` dataset is thus a collection of objects with type:

$$\texttt{COP} : \mathit{Bag}\,(\langle\, \texttt{cname} : \mathit{string}, \texttt{corders} : \mathit{Bag}\,(\langle\, \texttt{odate} : \mathit{date}, \texttt{oparts} : \mathit{Bag}\,(\langle \texttt{pid} : \mathit{int}, \texttt{qty} : \mathit{real} \rangle)\,\rangle)\,\rangle).$$

Figure 1.1 presents an example of the `COP` dataset, distributed across worker nodes using the top-level partitioning strategy. The tuples $\langle$`cname` : ...$\rangle$ of `COP` are treated as a single unit of information, with inner nested collections `corders` and `oparts` contained within it.

The distribution of the `Part` dataset is also presented in Figure 1.1. The `Part` dataset stores the IDs, name, and price of each part and has no nesting. The `Part` dataset is thus a collection of objects with type:

$$\texttt{Part} : \mathit{Bag}\,(\langle \texttt{pid} : \mathit{int}, \texttt{pname} : \mathit{string}, \texttt{price} : \mathit{real} \rangle).$$

Consider now a high-level collection program, `Totals`, that returns for each customer and for each of their orders, the total amount spent per part name. This requires joining `COP` with `Part` on `pid` to get the name and price of each part and then summing up the costs per part name. *Nested relational calculus* [8, 9] (NRC) is a declarative language for operating on and returning nested collections that can express this analysis:

```
1    Totals ⇐
2    for cop in COP union
3      {⟨ cname := cop.cname,
4         corders :=
5           for co in cop.corders union
6             {⟨ odate := co.odate,
7   Q           oparts := sumBy^total_pname(
8                            for op in co.oparts union
9      Q_corders               for p in Part union
10                               if op.pid == p.pid &&
11                                  co.odate == "01 : 03 : 22" then
12                     Q_oparts     {⟨ pname := p.pname,
13                                     total := op.qty * p.price ⟩})⟩}⟩}
```

The `Totals` program navigates to `oparts` in `COP` (lines 1-6), joins each bag with `Part` (line 9), computes the amount spent per part (line 11), and aggregates using `sumBy` which sums up the total amount spent for each distinct part name (line 6). The `Totals` program returns a collection of objects with output type:

$$\texttt{Totals} : Bag\,(\langle\, \texttt{cname} : string,\ \texttt{corders} : Bag\,(\langle\, \texttt{odate} : date, \texttt{oparts} : Bag\,(\langle \texttt{pname} : string,\ \texttt{total} : real\rangle)\,\rangle)\,\rangle).$$

There are several challenges associated with processing nested collections on top of distributed processing systems, related to both programming and performance. These challenges are outlined next.

1.1 Challenge 1: Programming mismatch

Programming languages, such as Java and Scala, provide collection APIs for transforming local collections. Large-scale processing frameworks provide similar APIs that execute batch operations in parallel over distributed partitions of data. The Spark API, for example, extends the Scala collection API to operate on distributed collections. Some Spark primitives are inherited directly from the Scala API, which are then applied locally at each data partition, such as filtering and mapping. The `filter` primitive applies selection returning only tuples that satisfy a boolean condition. The `map` operation defines a transformation on a collection. Similarly, `flatMap` performs a transformation on a nested collection to return a flattened result. Unlike local collection APIs, Spark provides a `join` primitive that merges tuples from two distributed collections based on a top-level key attribute. Operations that involve keys, like join, maintain a *key-based partitioning* guarantee that all values associated with a unique key are moved to the same partition. In contrast to the filter and map primitives, the join operation shuffles data across worker nodes. Once all values are in their key-based location the join is performed locally.

One may attempt to translate the association between `COP` and `Part` at lines 7-9 of the `Totals` program using the familiar filter and map operations:

```
COP.flatMap(cop => cop.corders.flatMap(co =>
  co.oparts.map( op => (op, Part.filter(p => p.pid == op.pid)))))
```

This program will be rejected from Spark since a distributed collection cannot be referenced within the transformation of a collection local to one worker node, ie. the

map on the local collection `co.oparts` attempts to transform the distributed collection `Part`. One attempt to solve this is to duplicate `Part` on each worker node using the `broadcast` primitive. After broadcasting, `Part` is a collection local to each worker node and the program above can be executed. However, the broadcast solution does not take full advantage of the distributed framework and the duplication of data across worker nodes can be prohibitively expensive.

A more efficient option is to rewrite the `Totals` program to use an explicit join operator on `pid`. Since the join primitive works on top-level key attributes, this requires flattening `COP` to bring `pid` values nested within `oparts` to the top-level. A natural way to flatten `COP` is to pair each `cname` with every tuple from `corders` and subsequently with every tuple from `oparts`. One attempt to return flat (`cname`, `odate`, `pid`, `qty`) tuples in Spark is:

```
COP.flatMap(cop => cop.corders.flatMap(co =>
  co.oparts.map( op => (cop.cname, co.odate, op.pid, op.qty))))
```

The flattened output materialized from this program is presented in Figure 1.2. This result has lost data and encodes incomplete information that can lead to an incorrect result. The first issue is that the flattened tuples omit customers with no orders, i.e. `customer2` tuple with empty `corders`, leading to data loss and fewer tuples in the result. To avoid this, `NULL` values must be inserted in place of empty inner collections. Second, the flattened result has lost all information to properly regroup leading to more data loss – ie. grouping the flattened result by `cname` and `odate` will have only one tuple in the output instead of two. To correctly associate nested values to top-level attributes, tuples must contain unique identifiers; for example, instead of grouping by `cname` and `odate` the grouping will be (`copID`, `cname`, `coID`, `odate`), where `copID` uniquely identifies each `cname` tuple and `coID` uniquely identifies each `corders` tuple. The handling of these `NULL` values and unique identifiers complicates downstream computation. Manually rewriting a program to work over the flattened representation while preserving its correctness is non-trivial. Further, flattening procedures can lead to exponential blowups in data size and contribute to data skew, which is related to the performance issues in the next two challenges.

```
{
 (cname: "customer1", odate: "01:03:22", pid: 1, qty: 5),
 (cname: "customer1", odate: "01:03:22", pid: 2, qty: 10),
 (cname: "customer1", odate: "01:03:22", pid: 3, qty: 2),
 (cname: "customer1", odate: "01:03:22", pid: 1, qty: 5),
 (cname: "customer1", odate: "01:03:22", pid: 2, qty: 10),
 (cname: "customer1", odate: "01:03:22", pid: 3, qty: 2)
}
```

Figure 1.2: Flattened `COP` resulting from example Spark program.

1.2 Challenge 2: Partitioning Nested Data

Since tuples are distributed as a single unit of information that contains all nested inner collections, the default top-level partitioning strategy of large-scale processing frameworks scales poorly for nested data. Figure 1.1 presents the top-level distribution of `COP` tuples across worker nodes. This partitioning limits parallelism of nested data, particularly with few top-level tuples and/or large inner collections. For example, worker 1 maintains the majority of the `COP` dataset. If the dataset contained only these two tuples then there are several worker nodes with empty data partitions. The varying sized inner collections leads to poor cluster utilization since only one worker is performing the majority of the work.

To cope with limited parallelism, the only option is to flatten nested data and redistribute across worker nodes. There are two main issues with this. First, the flattened representation leads to blowups in data size, which increases the load of data shuffling when redistributing. Second, flattening occurs locally at each node so varying sized inner collections can lead to partition imbalances that could completely overwhelm the resources on a worker node. In addition, flattening collections leads to data duplication and redundancy in computation. Prior to flattening, Figure 1.1 has one tuple per worker. After flattening, worker 1 contains the majority of the `COP` dataset with many duplicated tuples and will take longer to perform downstream operations. In the worst case, the initial flattening operation could crash the program even before repartitioning can occur. Representation of nested intermediates and outputs from transformations is likewise an issue. The flattened representation of nested data at any stage of the analysis can result in significant inefficiency in space usage. Exploration of succinctness and efficiency for different representations of nested output has not been the topic of significant study.

1.3 Challenge 3: Data Skew

Last but not least, data *skew* can significantly degrade performance of large-scale data processing, even when dealing with flat data only. Skew relates to the load imbalances across worker nodes, which causes some workers to perform significantly more work than others and prolongs runtimes.

The partitioning of nested data discussed so far is one contributor to skew. Inner collections may have skewed cardinalities – e.g., only a few customers can have corresponding orders information, or certain customers have more orders than others. Further, the standard top-level partitioning places each inner collection entirely on a single worker node. Programmers must ensure that inner collections do not grow large enough to cause disk spill or crash worker nodes due to insufficient memory.

While nested data exacerbates the problems of load imbalance, skew is a problem regardless of nesting. Another main contributor of skew is the key-based partitioning guarantee that works at top-level. In the worst case, all values could be associated to a single key and the whole dataset will be moved to a single node. Under these conditions distribution is completely lost. In general, skew-related issues can be difficult to diagnose and mitigate.

1.4 Existing Solutions

Prior work has addressed some of these challenges. NRC [8, 9] and monad comprehension calculus [10] have been previously proposed for transforming and operating on nested collections in centralized settings. High-level scripting languages such as Pig Latin [11] and Scope [12] ease the programming of distributed data processing systems. Apache Hive [13], F1 [14, 15], Dremel [7], and BigQuery provide SQL-like languages with extensions for querying nested structured data at scale. The for-comprehension syntax of Emma [16] and the SQL-like syntax of DIQL [17] provide DSLs that are deeply-embedded in Scala and target several distributed processing frameworks. An NRC implementation that targets distributed processing platforms and works with aggregation directly has not previously been considered. Further, all these solutions handle nested data with flattening solutions.

Early work on nested data [9, 18] proposed a flattening approach in which a nested collection is converted into a single collection containing both top-level attributes, lower-level attributes, and identifiers encoding inner collections - referred to here as flattened representations. Later work considered a normalized representation in which inner collections are stored separately, known as *shredding.* In this processing model, nested collections are encoded as a set of flat collections and nested collection queries are translated into relational queries. Shredding transformations that rewrite queries over the encoded structure have been explored previously in centralized settings. [19] proposed a *query shredding* transformation which converts nested relational queries to SQL queries that can be run on top of a commercial relational processor. While this is a starting point for the application of shredding to distributed execution, their translations are directly to SQL and do not provide flexibility for compiling to an arbitrary target, or applying additional optimizations at the level of a query plan. A shredded data representation and query transformation was proposed in [20], which focused on incremental evaluation. This work proved complexity, but did not provide an implementation or consider aggregation.

Skew-handling techniques have been previously explored that focuses on the identification of problematic, or *heavy*, keys and the management of their values [21, 22, 23]. Though skew can be worse when dealing with nested data, skew-handling has not been specifically explored in the context of nested data processing. There are efforts to identify long running workers, such as the speculative execution used in LATE [24] and Dolly [25]. These solutions are based on machine-level performance and are not focused on solving issues related to data distribution beyond top-level. In general, alleviating the performance problems caused by skew and manual flattening nested collections remains an open problem.

1.5 Proposed Solutions

To address the programming and performance challenges discussed above and achieve scalable processing of nested data, this research advocates an approach that relies on three key aspects: query compilation, query and data shredding, and skew-resilient query processing. The work extends NRC with key-based aggregation and adapts these extensions to shredding techniques in a compilation framework that targets dis-

tributed execution. Additional performance issues related to skew are automatically handled at runtime by maintaining the distribution of values associated to heavy keys. These contributions are outlined below.

1.5.1 Compilation of NRC Queries with Aggregation

To relieve programmers from manually rewriting programs for scalable execution (the programming mismatch challenge), this research proposes a compilation framework that automatically restructures programs to distributed settings. Users submit NRC queries to the framework, which are then compiled into a program defined in a distributed collection API. The NRC proposed in this research is akin to that existing in literature, but extended with the key-based aggregation operations that are commonplace in distributed collection APIs. Throughout compilation, the framework applies optimizations that are often overlooked by programmers such as column pruning, selection pushdown, and pushing nesting and aggregation operators past joins. The compilation framework thus transforms declarative NRC programs with aggregation into plans that are executable on distributed frameworks.

1.5.2 Query and Data Shredding

To achieve parallelism beyond top-level records (the nested data partitioning challenge), the compilation framework enables processing over *shredded* data. This thesis extends [20] for NRC with aggregation and integrates shredding into a compilation framework. The framework provides shredding as an optimized compilation route that encodes nested collections as a set of flat collections and translates nested collection queries into a set of queries that operate on those flat collections to return flat outputs. Users never interact with the shredding directly, but instead describe operations over nested data in NRC and the framework performs shredding in the backend.

The motivation for using shredding in distributed settings is twofold. First, a shredded representation of nested data enables full parallel processing of large nested collections and their even distribution among worker nodes. Second, the shredded representation is more *succinct* than the native representation which optimizes the communication costs of distributed execution. In the setting of complex sequences of nested-to-nested

data transformations, shredding does not need to reconstruct intermediate results. In addition, shredded outputs can share identifiers for inner bags that could be duplicated across objects. The shredded representation also provides more opportunities for reduction of data through aggregation, since such operations on the source object can be *localized* to only the necessary shredded component. The nested output can be later restored by joining together the computed flat collections. Otherwise, the shredded output can serve as input to another constituent query in a pipeline, presenting more opportunities for data sharing and reduction by aggregation.

A key motivation for this work is *analytics pipelines*, which typically consist of a sequence of transformations with the output of one transformation contributing to the input of the next. Although the final output consumed by an end user or application may be flat, the intermediate nested outputs may still be important in themselves: either because the intermediate outputs are used in multiple follow-up transformations, or because the pipeline is expanded and modified as the data is explored. The interest is thus in programs consisting of sequence of queries, where both inputs and outputs may be nested.

1.5.3 Automated Skew-Handling

The shredded representation naturally mitigates issues related to skew that arise from nested data. To handle additional data skew from key-based operations, the framework uses sampling techniques to automatically identify problematic keys and properly distribute their associated values at runtime. This process, referred to as *skew-resilient* query processing, extends the skew techniques of [23] in the setting of nested data with aggregation. Different evaluation plans are used for processing skewed and non-skewed portions of data. Plans are transparently rewritten to avoid partitioning collections on values that appear frequently in the data, and thus prevent the overloading of any of the worker nodes with significantly more data. This processing strategy deals with data skew appearing at the top level of distributed collections and, when coupled with the shredding transformation, effectively handles data skew present in inner collections.

1.5.4 Additional Contributions

This work combines the three main contributions above into a compilation framework called TraNCE (TRAnsforming Nested Collections Efficiently) that supports user-defined functions (UDFs) and provides an interactive, web-based frontend. UDFs have not previously been explored with shredding techniques. The work also provides a novel, nested query benchmark based on the TPC-H schema that focuses on scaling to deeper levels of nesting, large inner collections, and increasing amounts of skew. To evaluate the system using real-world datasets and analytical workloads, a biomedical benchmark is also provided. A case study for biomedical classification tasks is included, focusing on UDF optimization for feature selection workloads that also considers model performance.

1.6 Thesis-Related Collaborations and Publications

This thesis incorporates material from three papers, along with some material in preparation for submission. All work discussed here is in collaboration with Michael Benedikt and Milos Nikolic. The main body of work that details the majority of the technical aspects of the compilation framework is presented in [26], where the collaborators included also Amir Shaikhha. Additional material was made available in the arxiv paper [27]. The front end of our system was described in the demonstration paper [28]: Brandon Moore was also a collaborator on this project. The biomedical motivation, benchmark, and the extensions to the framework to support biomedical analyses are described in the journal paper [29]. This included the work of Yao Shi's masters thesis, which was done under my supervision. The extensions for user-defined functions, feature selection workloads, and associated filters were the basis of Pavlos Piperis' masters project, also done under my supervision.

1.7 Thesis Organization

The thesis begins by revisiting several strands of previous work that inspired the components of the compilation framework in Chapter 2. The foundation of our compila-

tion framework is presented here in the context of our newly proposed NRC source language. The extensions of NRC with aggregation are then presented in Chapter 3, followed by the shredding extensions in Chapter 4. Skew-resilience and additional optimizations are presented in Chapter 5. Chapter 6 presents the architecture of the framework. A systematic evaluation of the framework and competitors is provided in Chapter 7, using our nested TPC-H benchmark. The remaining chapters focus on the biomedical applications and include further background on multi-omics biomedical analyses. The thesis concludes with a discussion on limitations and future work. The framework implementation and benchmarks are available at [30].

Chapter 2

Foundations of the Framework and Background

This thesis presents TraNCE, a compilation framework designed for transforming nested collections on top of distributed processing systems. To the best of our knowledge, this is the first framework to integrate query compilation, shredding, and skew-handling techniques into a single framework that targets distributed execution. TraNCE ties together several strands of previous work, which are revisited throughout this chapter using our newly proposed syntax for NRC [8, 9] and related intermediate languages. Each section discusses the background techniques that inspired the compilation components, along with the relevant motivation for the framework.

To solve the programming mismatch, TraNCE uses query compilation to produce a distributed application from an input NRC program. The nested data model and base syntax for NRC are presented in Section 2.1 and 2.2. Source queries are translated into plans designed for bulk execution, which is based on a plan translation procedure from [10]. The plan language and plan translation are presented in Section 2.3 and 2.4. To optimize nested data processing, the framework also provides a shredding module that is based on previous techniques [19, 20]. The shredded representation and the basis of the shredding transformation are presented in Section 2.5, which adapts [20] to work with the nested data model and our NRC syntax. Finally, query plans in TraNCE undergo dynamic optimizations that overcome skew-related issues at run-

time. These skew-handling techniques are based on previous techniques for identifying and handling problematic keys, discussed in Section 2.7. The chapter also provides an overview of distributed processing frameworks, nested data processing implementations, and UDFs.

2.1 Nested Data Model

The nested data model, also called the "complex object data model" supports collections of tuples whose values may also be collections. This is a strongly-typed model that requires collection types to be homogeneous. The model builds up nested collection types from the basic scalar types (integer, string, etc.), tuple type $\langle a_1 : T_1 \ldots a_n : T_n \rangle$, and bag type $Bag(F)$. For ease of presentation, bag content F is restricted to be either a tuple type or a scalar type.

$$
\begin{aligned}
T &::= S \mid C \\
F &::= \langle a_1 : T, \ldots, a_n : T \rangle \mid S \\
C &::= Bag(F) && \textit{– Collection Type} \\
S &::= int \mid real \mid string \mid bool \mid date && \textit{– Scalar Type}
\end{aligned}
$$

Collections are represented as bags with schemas presented as `name`:Bag. As an example, consider `R` and `S` which are collections of tuples with the following types:

$$
\begin{aligned}
&\texttt{R} : Bag(\langle \texttt{a} : int, \texttt{b} : int, \texttt{c} : Bag(\langle \texttt{d} : int, \texttt{e} : Bag(\langle \texttt{f} : int \rangle)\rangle)\rangle) \\
&\texttt{S} : Bag(\langle \texttt{a} : int, \texttt{b} : int \rangle)
\end{aligned}
$$

`R` contains a nested collection at attribute `c`, which also contains a nested collection at attribute `e`. `S` is a flat collection with no nested attributes. Example datasets of both `R` and `S` are provided below:

$$
\begin{aligned}
&\texttt{R} : \{\langle \texttt{a} : 1, \texttt{b} : 1, \texttt{c} : \{\langle \texttt{d} : 1, \texttt{e} : \{\langle \texttt{f} : 4\rangle\}\rangle, \langle \texttt{d} : 2, \texttt{e} : \{\langle \texttt{f} : 5\rangle\}\rangle\}\rangle,\\
&\quad \langle \texttt{a} : 1, \texttt{b} : 2, \texttt{c} : \{\langle \texttt{d} : 2, \texttt{e} : \{\langle \texttt{f} : 4\rangle\}\rangle\}\rangle,\\
&\quad \langle \texttt{a} : 2, \texttt{b} : 3, \texttt{c} : \{\langle \texttt{d} : 3, \texttt{e} : \{\langle\rangle\}\rangle\}\rangle\}\\
&\texttt{S} : \{\langle \texttt{a} : 1, \texttt{b} : 5\rangle, \langle \texttt{a} : 2, \texttt{b} : 6\rangle\}
\end{aligned}
$$

Figure 1.1 also provides an example dataset COP that satisfies the nested data model.

The proposal of the nested data model dates back to the late seventies [31] when the first normal form was relaxed in an effort to support object-oriented applications. The field began to mature roughly a decade later as both the database and programming language communities were attempting to provide complex object support from virtually opposite directions.

Many data types used today are richer and less strict than the nested data model, such as XML, JSON, and Protocol Buffers. XML and JSON are semi-structured types that provide support for heterogeneous types, missing attributes, and ordering. Protocol Buffers use a structured schema and support missing attributes. The work of this thesis is specifically focused on the nested data model. Any additional issues specific to processing data in the richer model are out of scope and marked for future work.

2.2 Nested Relational Calculus

Nested relational calculus (NRC) is a declarative query language for operating on and returning nested collections [32, 18]. The following grammar describes one standard syntax for NRC from the literature, which this thesis builds upon:

$$
\begin{aligned}
e &::= \emptyset_{Bag(F)} \mid \{e\}\\
&\quad \mid var \mid e.a \mid \langle a_1 := e, \ldots, a_n := e\rangle\\
&\quad \mid \texttt{for}\ var\ \texttt{in}\ e\ \texttt{union}\ e \mid e \uplus e\\
&\quad \mid \texttt{if}\ cond\ \texttt{then}\ e\ \texttt{else}\ e\\
&\quad \mid \texttt{let}\ var := e\ \texttt{in}\ e \mid e\ PrimOp\ e\\
cond &::= e\ RelOp\ e \mid \neg cond \mid cond\ BoolOp\ cond
\end{aligned}
$$

RelOp is a comparison operator on scalar values (e.g., $==$, $\leq$), *PrimOp* is a primitive function on scalar values (e.g., $+$, $*$), and *BoolOp* is a boolean operator (e.g., &&, ||). A simple term is either a variable or an expression $e.a$, where a is an attribute. Variables can either be free variables of the expression representing input objects or variables introduced in `for` or `let` constructs. $\{e\}$ takes the expression e and returns a singleton bag. $\emptyset_{Bag(F)}$ returns an empty bag of type $Bag(F)$.

The semantics of NRC are provided, letting p range over environments mapping variables to values, such that a non-empty environment $p[var \to v]$ denotes the extension of p with var bound to v. The notation used follows [19] and [20].

$$\begin{aligned}
[\![\emptyset_{Bag(F)}]\!]_p &= \{\} \\
[\![\{e\}]\!]_p &= \{[\![e]\!]_p\} \\
[\![var]\!]_p &= p(var) \\
[\![e.a]\!]_p &= [\![e]\!]_p.a \\
[\![\langle a_1 := e_1, \ldots, a_n := e_2 \rangle]\!]_p &= \langle a_1 := [\![e_1]\!]_p, \ldots, a_n := [\![e_2]\!]_p \rangle \\
[\![\texttt{for } var \texttt{ in } e_1 \texttt{ union } e_2]\!]_p &= \biguplus_{v \in [\![e_1]\!]_p} [\![e_2]\!]_{p[var:=v]} \\
[\![e_1 \uplus e_2]\!]_p &= [\![e_1]\!]_p \uplus [\![e_2]\!]_p \\
[\![e_1 \; PrimOp \; e_2]\!]_p &= [\![e_1]\!]_p \; PrimOp \; [\![e_2]\!]_p \\
[\![\texttt{if } cond \texttt{ then } e_1 \texttt{ else } e_2]\!]_p &= \begin{cases} [\![e_1]\!]_p \; if [\![cond]\!]_p = true \\ [\![e_2]\!]_p \; if [\![cond]\!]_p = false \end{cases} \\
[\![\texttt{let } var := e_1 \texttt{ in } e_2]\!]_p &= [\![e_2]\!]_{p[var:=[\![e_1]\!]_p]}
\end{aligned}$$

As an example, consider the NRC expression `Ex1` that works over collections `R` and `S`:

```
for r in R union
  for s in S union
    if r.a == s.a then
      {⟨a := r.a, b := r.b ∗ s.b⟩}
```

The materialized results of `Ex1` is $\{\langle$ `a`: 1, `b`: 5 $\rangle$, $\langle$ `a`: 1, `b`: 10 $\rangle$, $\langle$ `a`: 2, `b`: 18 $\rangle\}$.

The following expression `Ex2` provides an example of an expression that has a nested output type:

```
for r in R union
  {⟨a := r.a, g :=
    for c in r.c union
      for s in S union
        if c.d == s.a then
          {⟨d := c.d, b := r.b ∗ s.b⟩}
```

The materialized results of `Ex2` is $\{\langle$ a: 1, g: $\{\langle$ d: 1, b: 5 $\rangle$, $\langle$ d: 2, b: 6 $\rangle\}\rangle$, $\langle$ a: 1, g: $\{\langle$ d: 2, b: 12 $\rangle\}\rangle$, $\langle$ a: 2, g: $\{\}$ $\rangle\}$. Note that the aggregation used in the `Totals` program from the introduction is actually not provided in NRC. A main focus of this thesis is to extend NRC with such key-based aggregation.

2.2.1 A Brief History

In the past, nested data processing focused on the technical challenges of mapping language-level objects to relational systems. This was known as the *impedance mismatch* and was approached from a variety of angles. Some attempted complex object support with persistent memory extensions, others focused on extending the relational model to support object types [33]. Object Oriented Database (OODB) Systems [33] bridged these approaches. OODBs lacked a declarative language and instead used navigational approaches for accessing nested tuples with inter-object references, which limited opportunities for optimization. By the 2000s, the Object Database Management Group presented the object data standard (ODMB) [34] and associated declarative Object Query Language (OQL) [35]; however, OQL was a simple language that still had limited support for optimization. Object-relational languages were also proposed that extended the relational model and SQL, supporting complex objects in SQL3 and object standards in SQL:1999 [36].

Many declarative languages for the nested model, such as NRC, were influenced by programming languages. While nested relational algebra was initially made by extending relational algebra with unnest and nest operations, the use of the monad calculus simplified procedural approaches by introducing a map construct and first-order function definitions [32]. Inspired by application programming tasks that operate over collection primitives, a more convenient *comprehension-style* syntax for NRC was provided in the language CPL [9], which served as the basis for the Kleisli system [37].

2.2.2 Comprehension-Style Syntax

The comprehension-style syntax is a syntactic abbreviation of NRC, where `for` constructions are presented as comprehensions: every occurrence of `for` *var* `in` e `union` e is treated as $\{e|var \ \leftarrow \ e, \ldots, e\}$. The left side of the comprehension (head) is built

up from the iterations in the right side of the comprehension (body), defined via qualifiers. The body can also contain conditional expressions that are referred to as predicates. We refer to NRC with comprehension-syntax as NRC^+.

As an example, the expression `Ex1` is equivalent to the following in NRC^+:

$$\{\{\langle\ \mathtt{a} := \mathtt{r.a},\ \mathtt{b} := \mathtt{r.b} * \mathtt{s.b}\ \rangle \mid \mathtt{s} \leftarrow \mathtt{S},\ \mathtt{r.a} == \mathtt{s.a}\ \} \mid \mathtt{r} \leftarrow \mathtt{R}\ \}.$$

Anything to the right of a | is considered the body with the head on the left-most side. The body contains $\mathtt{r} \leftarrow \mathtt{R}$ and $\mathtt{s} \leftarrow \mathtt{S}$ as qualifiers and $\mathtt{r.a} = \mathtt{s.a}$ as the only predicate. The head contains the tuple $\langle \mathtt{a} := \mathtt{r.a}, \mathtt{b} := \mathtt{s.b}\rangle$. This can also be expressed using a short-hand syntax [10], where the qualifier is first defined, followed by the remaining qualifiers $\overline{r}$ and predicates p in the comprehension body. The head expression is represented merely as e. The example comprehension in short-hand is: $\{e|\mathtt{r} \leftarrow \mathtt{R}, \overline{r}, p\}$; with e in-place of $\langle \mathtt{a} := \mathtt{r.a}, \mathtt{b} := \mathtt{s.b}\rangle$, $\overline{r}$ in-place of $\mathtt{s} \leftarrow \mathtt{S}$, and p in-place of $\mathtt{r.a} == \mathtt{s.a}$.

The NRC^+ syntax supports nested comprehensions, such that a comprehension can lie within the head of another comprehension. The comprehension associated to `Ex2` is an example of a nested comprehension:

$$\{\langle\ \mathtt{a} := \mathtt{r.a},\ \mathtt{g} := \{\{\langle\ \mathtt{d} := \mathtt{c.d},\ \mathtt{b} := \mathtt{r.b} * \mathtt{s.b}\ \rangle\ |\mathtt{s} \leftarrow \mathtt{S},\ \mathtt{c.d} == \mathtt{s.a}\}|\ \mathtt{c} \leftarrow \mathtt{r.c}\ \}\}\rangle \mid \mathtt{r} \leftarrow \mathtt{R}\ \}$$

The comprehension syntax of NRC^+ is accompanied with normalization rules [10] that are applied exhaustively to remove redundant comprehensions and reduce the amount of intermediate collections. Comprehension-syntax is considered more compositional than NRC, facilitating downstream optimizations such as the generation of query plans discussed in the next two sections. This thesis uses NRC^+ internally in the framework during several stages of compilation.

2.3 Plan Language

Queries can be translated to an algebraic form that represents the stages of query processing as more efficient bulk operations, referred to as a *query plan*. Plans are built up from the operators of a plan language, presented with semantics below. Each e is

an NRC^+ expression of type *bool* and X and Y of type *Bag*. An attribute is denoted with a and $\overline{a}$ a sequence of attributes $a_1, \ldots, a_k$. Each tuple x in X is short-hand for $\langle a_1 : x.a_1, \ldots, a_k : x.a_k \rangle$. Similarly, each tuple $\langle x, y \rangle$ for x in X and y in Y is short-hand for $\langle a_1 : x.a_1, \ldots, a_k : x.a_k, b_1 : y.b_1, \ldots, y : b_j \rangle$. The function ID produces a unique identifier given an input tuple.

$$
\begin{array}{lll}
\sigma_e \text{ X} & = & \{\texttt{if } e \texttt{ then } \{x\} \texttt{ else } \{\} \mid x \leftarrow X\} \\
\pi_{a_1,\ldots,a_k} X & = & \{\langle a_1 : x.a_1, \ldots, a_k : x.a_k \rangle \mid x \leftarrow X\} \\
X \bowtie_e Y & = & \{\texttt{if } e \texttt{ then } \{\langle x, y \rangle\} \texttt{ else } \{\} \mid x \leftarrow X, y \leftarrow Y\} \\
X ⟕_e Y & = & \{\texttt{if } e\ \{\langle x, y \rangle\} \texttt{ else } \{\langle x, \texttt{NULL} \rangle\} \mid x \leftarrow X, y \leftarrow Y\} \\
\mu^a X & = & \{\langle x, y \rangle \mid x \leftarrow X, y \leftarrow x.a\} \\
\overline{\mu}^a X & = & \{\texttt{if } x.a == \texttt{NULL}\ \{\langle x, \texttt{NULL} \rangle\} \texttt{ else } \{\langle x, y \rangle\} \mid x \leftarrow X, y \leftarrow x.a\} \\
\Gamma^{\uplus/\overline{v}}_{\overline{a}} X & = & \{\langle x.\overline{a},\ \{x'.\overline{v} \mid x' \leftarrow X, x.\overline{a} == x'.\overline{a}\ \} \rangle \mid x \leftarrow X\} \\
I & = & \{\langle x, \texttt{ID}(x) \rangle \mid x \leftarrow X\}
\end{array}
$$

The plan language includes selection σ, projection π, join $\bowtie$, and left outer join ⟕, as known from relational algebra. The unnest operator μ^a takes a nested bag with top-level bag-valued attribute a and pairs each tuple of the outer bag with each tuple of a, while projecting away a. The outer-unnest operator $\overline{\mu}^a$ is a variant of μ^a that performs the pairings of the unnest operator while also placing NULL values when a is an empty bag. The operator $\Gamma^{\uplus/\overline{v}}_{\overline{a}}$ defines grouping to produce inner nested collections that have projected $\overline{v}$ for each projection of $\overline{a}$; this is referred to as the nest operator. The Γ operator casts NULL values introduced by the outer operators to the empty bag. The index operator $I(e)$ appends a unique identifier to every tuple in the input e using the ID function.

2.4 Plan Translation

The process of translating an NRC^+ expression to a query plan is referred to as *comprehension to plan translation*, or plan translation for short. The goal of this translation is to represent the query in an algebraic form that facilitates code generation and database-style optimizations. For flat comprehensions, the plan translation only

involves the standard relational operators. For nested comprehensions, the plan translation involves a decorrelation procedure, known as *unnesting* [10], which removes any query nesting in a comprehension.

Plan translation works top-down on a comprehension, recursively applying the plan translation rules to build up an expression from the plan operators. The plan translation rules are provided in Figure 2.1 following [10] with a few minor adjustments. First, there is explicit use of the index operator I to replace the concept of *dot-equality.* The index operator appends a unique identifier to every input tuple, as defined in Section 2.3. These unique identifiers are essential for preserving the multiplicities of parent-levels. Second, the environment tracks attribute sets rather than variable sets. The tracking of attributes is a better intermediate representation for the bulk, column-based transformations used in distributed collection APIs.

Each plan translation call builds up a subplan E and returns the remaining comprehension within the plan translation environment, `PTrans`: $\texttt{PTrans}(\{e|v \leftarrow X, \overline{r}, p\})^{u}_{w}\ E$. The remaining comprehension is represented as $\{e|v \leftarrow X, \overline{r}, p\}$, w is the set of attributes (α) from the current environment, and u is the subset of those attributes that originate from the parent environment.

Base inputs (X) can be an input data source or materialized from an assignment upstream in the program. The index operator I is applied to every collection associated to a qualifier. The set of attributes for a variable v is denoted α^{v} and α^{v}_{I} after the index operator is applied. Predicates are split based on those involving a variable v, $p[v]$ and those involving all other variables $p[\overline{v}]$. Expressions in the head are similarly applied, ie. $e[v]$. All plan translation rules are meaning preserving based on Corollary 1 and TH2 from [10]: $\texttt{PTrans}(\{\ e \mid \overline{r}\ \}])^{()}_{()}() = \{\ e \mid \overline{r}\ \}$ inherited from $\texttt{PTrans}(\{\ e \mid \overline{r}\ \})^{()}_{w}E = \{\ e \mid w \leftarrow E, \overline{r}\ \}$.

Translation starts from the outermost level of a comprehension, applying the rules U1 - U8 to recursively build up a subplan E from a comprehension $\{e|\overline{r}\}$. The base case (U1) is when all components of the environment (u, w, E) are empty. Rules U2 - U7 will be triggered depending on the value of u. Joins (U2, U3) are created when the subplan is non-empty and a qualifier is present, capturing the relevant join-condition predicates. Unnest operators (U4, U5) are created when the qualifier is referencing a nested collection. The outer-variants of these operators are produced when u is non-

Plan translation rules		
U1 Selection produced when u, w and E are empty		
$\texttt{PTrans}(\{e \vert v \leftarrow X, \overline{r}, p\})^{()}_{()}()$	$=$	$\texttt{PTrans}(\{e \vert \overline{r}, p[\overline{v}]\})^{()}_{(\alpha^v_I)}(\sigma_{p[v]} I(X))$
U2 Join when u is empty and w and E are non-empty		
$\texttt{PTrans}(\{e \vert v \leftarrow X, \overline{r}, p\})^{()}_{w} E$	$=$	$\texttt{PTrans}(\{e \vert \overline{r}, p[\overline{(w,v)}]\})^{()}_{(\alpha^w_I, \alpha^v_I)}(E \bowtie_{p[(w,v)]} (\sigma_{p[v]} I(X)))$
U3 OuterJoin when u, w and E are non-empty		
$\texttt{PTrans}(\{e \vert v \leftarrow X, \overline{r}, p\})^{u}_{w} E$	$=$	$\texttt{PTrans}(\{e \vert \overline{r}, p[\overline{(w,v)}]\})^{u}_{(\alpha^w_I, \alpha^v_I)}(E ⟕_{p[(w,v)]} (\sigma_{p[v]} I(X)))$
U4 Unnest when u is empty and w and E are non-empty		
$\texttt{PTrans}(\{e \vert v \leftarrow path, \overline{r}, p\})^{()}_{w} E$	$=$	$\texttt{PTrans}(\{e \vert \overline{r}, p[\overline{v}]\})^{()}_{(\alpha^w_I, \alpha^v_I)} I(\mu^{path}_{p[v]} E)$
U5 OuterUnnest when u, w and E are non-empty		
$\texttt{PTrans}(\{e \vert v \leftarrow path, \overline{r}, p\})^{u}_{w} E$	$=$	$\texttt{PTrans}(\{e \vert \overline{r}, p[\overline{v}]\})^{u}_{(\alpha^w_I, \alpha^v_I)} I(\mu^{path}_{p[v]} E)$
U6 Projection u is empty, w and E are non-empty		
$\texttt{PTrans}(\{e \vert p\})^{()}_{w} E$	$=$	$\pi_e(\sigma_p(E))$
U7 Nest u, w and E are non-empty		
$\texttt{PTrans}(\{e \vert p\})^{u}_{w} E$	$=$	$\Gamma^{\uplus/e[u]}_{e[w-u]}(\sigma_p(E))$
U8 Handle comprehensions in the head when w and E are non-empty		
$\texttt{PTrans}(\{f(\{e \vert \overline{r}\}) \vert p\})^{u}_{w} E$	$=$	$\texttt{PTrans}(\{f(v) \vert p\})^{u}_{(\alpha^w_I, \alpha^v_I)} I(\texttt{PTrans}(\{e \vert \overline{r}\})^{\alpha^w_I}_{\alpha^w_I} E)$

Figure 2.1: Plan translation rules translating NRC^+ to plan language operators.

empty; the use of outer operators prevents data loss with the insertion of nulls for any missing values. Recall that u is non-empty when attributes have been collected from a parent level, ie. the algorithm has entered a new nesting level (U8). When the comprehension body is empty, operators are placed to terminate the subplan for that level. The nest operator (U7) terminates a subplan when building up a non-root level (u is non-empty). The algorithm terminates with the projection operator (U6) when u is empty and the comprehension body is empty.

The example below walks through the plan translation on the NRC^+ `Ex2` expression. The algorithm works outside-in on the comprehension applying the rules from Figure 2.1. Short-hand comprehensions $\{e \vert \overline{r}\}$ are presented as truncated views of lower-level

comprehensions; these comprehensions and the components of the plan translation environment are tagged with level identifiers to distinguish recursive calls at various levels. At each call, a variable v is created with the type of the current attribute set, which replaces any variable reference in the current level comprehension. Bag-typed attributes later handled by unnest operators are omitted from the attribute sets.

Step 0: The plan translation starts with an empty u, w, and E.

$^{0}\texttt{PTrans}(\{\langle \texttt{a} := \texttt{r.a},\ \texttt{g} := \{e_0 | \overline{r_0}\}\rangle | \texttt{r} \leftarrow \texttt{R}\})^{()}_{()}\{()\}$

Step 1: U1 is the first rule applied, capturing the R qualifier and appending a unique rID. The variable v_0 replaces r in the projection on a.

$^{0}\texttt{PTrans}(\{\langle \texttt{a} := v_0\texttt{.a},\ \texttt{g} := \{e_0 | \overline{r_0}\}\rangle | \}\})^{()}_{w_0} I(\texttt{R})$

$w_0 = \{\texttt{rID}, \texttt{a}, \texttt{b}\}$

Step 2: With no more qualifiers in the body, U8 is triggered to handle the comprehension $\{e_0 | \overline{r_0}\}$ in the head. The current environment w is passed in as the parent attributes u.

$^{0}\texttt{PTrans}(\{\langle \texttt{a} := v_0\texttt{.a},\ \texttt{g} :=:= v_0\texttt{.g}\rangle | \}\})^{()}_{w_0,\texttt{g}}$
$^{1}\texttt{PTrans}(\{\langle \texttt{d} := \texttt{c.d}, \texttt{b} := \texttt{r.b} * \texttt{s.b}\rangle | \texttt{c} \leftarrow \texttt{r.c}, \texttt{s} \leftarrow \texttt{S},\ \texttt{c.d} == \texttt{s.a}\})^{w_0}_{w_0} I(\texttt{R})$

$w_0 = \{\texttt{rID}, \texttt{a}, \texttt{b}\}$

Step 3: U5 is triggered from a non-empty u and a path in the body, producing an outer-unnest on c wrapped in an index.

$^{0}\texttt{PTrans}(\{\langle \texttt{a} := v_0\texttt{.a},\ \texttt{g} := v_0\texttt{.g}\rangle | \}\})^{()}_{w_0,\texttt{g}}$
$^{1}\texttt{PTrans}(\{\langle \texttt{d} := \texttt{c.d}, \texttt{b} := \texttt{r.b} * \texttt{s.b}\rangle | \texttt{s} \leftarrow \texttt{S},\ \texttt{c.d} == \texttt{s.a}\})^{u_1}_{w_1} I(\neg\!\!\mu_{\texttt{c}}\ (I(\texttt{R})))$
$u_1 = \{\texttt{rID}, \texttt{a}, \texttt{b}\}$
$w_1 = u_1 \cup \{\texttt{cID}, \texttt{d}\}$

Step 4: The next qualifier with a non-empty u produces an outer-join with the indexed $\mathtt{S}$.

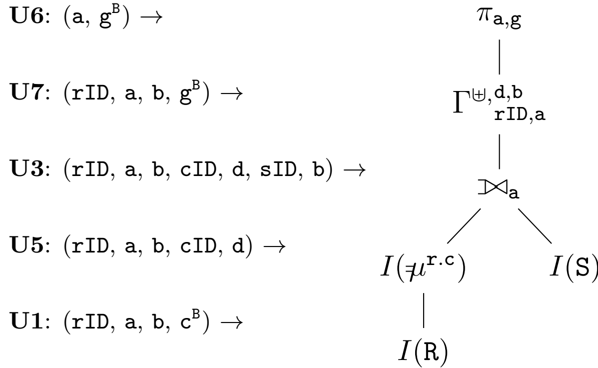
$$^{0}\mathtt{PTrans}(\{\langle \mathtt{a} := v_0.\mathtt{a},\ \mathtt{g} := v_0.\mathtt{g}\rangle|\})^{()}_{w_0,\mathtt{g}}$$
$$^{1}\mathtt{PTrans}(\{\langle \mathtt{d} := \mathtt{c.d}, \mathtt{b} := \mathtt{r.b} * \mathtt{s.b}\rangle|\})^{w_1}_{w_1} I(\mu_{\mathtt{c}}\ (I(\mathtt{R}))) ⟕_{\mathtt{d=a}} I(\mathtt{S})$$

$$u_1 = \{\mathtt{rID}, \mathtt{a}, \mathtt{b}\}$$
$$w_1 = u_1 \cup \{\mathtt{cID}, \mathtt{d}, \mathtt{sID}, \mathtt{b}\}$$

Step 5: The first-level is finalized with U7 with u_1 non-empty.

$$^{0}\mathtt{PTrans}(\{\langle \mathtt{a} := v_0.\mathtt{a},\ \mathtt{g} := v_0.\mathtt{g}\rangle|\})^{()}_{w_0,\mathtt{g}}\ \Gamma^{\uplus/\langle \mathtt{d},\mathtt{b}\rangle}_{u_1}(I(\mu_{\mathtt{c}}\ (I(\mathtt{R}))) ⟕_{\mathtt{d=a}} I(\mathtt{S}))$$

Step 6: The top-level plan translation environment is finalized with U6 and the algorithm terminates returning the plan below with output types. Bag types are annotated with ^{B}.



The above example presents how a comprehension can be translated into a query plan that describes bulk operations over nested data. In TraNCE, the query plan facilitates downstream optimizations and code generation. Our work adapts the plan translation procedure to work with NRC^+, aggregation extensions, and shredding.

2.5 Query and Data Shredding

To support more efficient processing, query languages for complex objects can operate on a variety of internal data representations. These representations must be coupled with a corresponding query transformation when the logical structure of the original nested representation is not preserved. One approach is to use *flattening*, which transforms a nested collection to a single flat collection containing top-level attributes, inner attributes and associated identifiers [18, 9]. A query over nested data is then transformed to an equivalent query over the flattened representation. An example of a flattened representation for `COP` was provided in Figure 1.2. The pairing of every top-level tuple with each inner tuple, however, can result in space inefficiencies. Later work proposed a normalized, *shredded* representation that stores inner collections separately, using identifiers to maintain the relationships from the nested structure. The process of producing these inner collections is referred to as *data shredding*, which transforms a nested collection into a set of flat inputs. The query that operates on nested data then undergoes a *query shredding* [19, 20] transformation to operate over the set of flat inputs. This thesis adapts previous shredding techniques to target distributed processing frameworks and support NRC with aggregation. The background on the shredded representation and associated transformations are presented next in the context of our NRC syntax.

2.5.1 Shredded Representation

The *shredded representation* encodes a nested bag b with attributes $a_1, \ldots, a_k$ into a top-level flat bag b^{F} where each a_i of bag type is replaced with a *Label* that acts as an identifier to the bag at a_i. The mappings to label identifiers and the corresponding flat bags are maintained in a dictionary tree b^{D}, which is a tuple of recursive dictionary definitions for all bag type a_i. A recursive dictionary tree includes attributes $a_i^{\mathtt{fun}}$ and $a_i^{\mathtt{child}}$ for each inner bag-valued a_i, where $a_i^{\mathtt{fun}}$ is of dictionary type and $a_i^{\mathtt{child}}$ denotes the dictionary tree for lower-level bags. Dictionaries produced from data shredding are termed *value dictionaries* since these encode actual values. Alternatively, dictionaries produced from query shredding are referred to as *dictionary expressions.*

$$
\begin{aligned}
T^{\mathrm{F}} &::= \mathit{Bag}(F) && \text{– \textit{Flat Collection}} \\
F &::= \langle a_1 : S, \ldots, a_n : S \rangle \mid S && \text{– \textit{Flat Tuple}} \\
S &::= \mathit{int} \mid \mathit{real} \mid \mathit{string} \mid \mathit{bool} \mid \mathit{date} \mid \mathit{Label} && \text{– \textit{Label Type}} \\
D &::= \mathit{Label} \to T^{\mathrm{F}} && \text{– \textit{Dictionary Type}} \\
T^{\mathrm{D}} &::= \langle \ldots, a_i : \langle a_i^{\texttt{fun}} : D, a_i^{\texttt{child}} : \{T^{\mathrm{D}}\} \rangle, \ldots \rangle && \text{– \textit{Recursive Dictionary}}
\end{aligned}
$$

The type of the shredded representation is formed via a transformation on the type T of bag b. The type T^{F} of b^{F} is always a flat bag type. For a tuple type $T = \langle a_1 : T_1, \ldots, a_n : T_n \rangle$, T^{F} is formed by replacing each attribute a_i of bag type with *Label*. The type T^{D} of b^{D} includes the $a_i^{\texttt{fun}}$ and $a_i^{\texttt{child}}$ attributes for each a_i of bag type. Similarly, if $T = \mathit{Bag}(T_1)$ then $T^{\mathrm{F}} = \mathit{Bag}(T_1^{\mathrm{F}})$ and $T^{\mathrm{D}} = T_1^{\mathrm{D}}$, and scalar types give $T^{\mathrm{F}} = T$ and $T^{\mathrm{D}} = \langle \rangle$. Empty dictionary definitions $\langle \rangle$ are omitted from the recursive dictionary tuple for clarity.

Consider the example dataset `R`, the shredded representation is:

$$
\begin{aligned}
&\texttt{R}^{\mathrm{F}} : \mathit{Bag}(\langle\ \texttt{a} : \mathit{int}, \texttt{b} : \mathit{int}, \texttt{c} : \mathit{Label}\ \rangle) \\
&\texttt{R}^{\mathrm{D}} : \langle \texttt{c} : \langle \texttt{c}^{\texttt{fun}} : \mathit{Label} \to \mathit{Bag}(\langle\ \texttt{d} : \mathit{int}, \texttt{e} : \mathit{Label}\ \rangle) \\
&\qquad \texttt{c}^{\texttt{child}} : \langle \langle \texttt{e} :\ \texttt{e}^{\texttt{fun}} : \langle \mathit{Label} \to \mathit{Bag}(\langle\ \texttt{f} : \mathit{int}\ \rangle), \texttt{e}^{\texttt{child}} : \mathit{Bag}(\langle \rangle) \rangle \rangle \rangle \rangle
\end{aligned}
$$

To simplify the presentation, attributes are dropped for recursive dictionary tuples that only contain one bag-typed attribute:

$$
\begin{aligned}
&\texttt{R}^{\mathrm{F}} : \mathit{Bag}(\langle\ \texttt{a} : \mathit{int}, \texttt{b} : \mathit{int}, \texttt{c} : \mathit{Label}\ \rangle) \\
&\texttt{R}^{\mathrm{D}} : \langle \texttt{c}^{\texttt{fun}} : \mathit{Label} \to \mathit{Bag}(\langle\ \texttt{d} : \mathit{int}, \texttt{e} : \mathit{Label}\ \rangle) \\
&\qquad \texttt{c}^{\texttt{child}} : \mathit{Bag}(\langle\ \texttt{e}^{\texttt{fun}} : \langle \mathit{Label} \to \mathit{Bag}(\langle\ \texttt{f} : \mathit{int}\ \rangle), \texttt{e}^{\texttt{child}} : \mathit{Bag}(\langle \rangle) \rangle \rangle \rangle
\end{aligned}
$$

Here, the first-level dictionary can be referred to by index as $\mathtt{R}^{\mathrm{D}}.\mathtt{c}^{\mathtt{fun}}$ and the second as $\mathtt{R}^{\mathrm{D}}.\mathtt{c}^{\mathtt{child}}.\mathtt{e}^{\mathtt{fun}}$.

The shredded representation that uses the recursive dictionary type is referred to as the *recursive* view. An alternative, *relational view*, of the shredded representation represents dictionaries as a collection of flat tuples that contain label attributes. In the relational view, each component acts as a table with labels as functional dependencies. Each dictionary component can be referred to by path index. An example of the relational view for R is:

$$\begin{aligned}
\mathtt{R}^{\mathrm{F}} &: \mathit{Bag}(\langle\ \mathtt{a} : \mathit{int}, \mathtt{b} : \mathit{int}, \mathtt{c} : \mathit{Label}\ \rangle) \\
\mathtt{R}^{D}_{\mathtt{c}} &: \mathit{Bag}(\langle\ \mathtt{lbl} : \mathit{Label}, \mathtt{d} : \mathit{int}, \mathtt{e} : \mathit{Label}\ \rangle)) \\
\mathtt{R}^{D}_{\mathtt{c_e}} &: \mathit{Bag}(\langle\ \mathtt{lbl} : \mathit{Label},\ \mathtt{f} : \mathit{int}\ \rangle))
\end{aligned}$$

Here, the first-level dictionary is referred to as $\mathtt{R}^{D}_{\mathtt{c}}$ and the lowest-level dictionary as $\mathtt{R}^{D}_{\mathtt{c_e}}$, where c_e is the path.

2.5.2 Data Shredding and Unshredding

Nested objects can be converted to their shredded representations and vice versa. A *value shredding function* VSHRED takes as input a nested object o and returns a top-level bag o^{F} and a dictionary tree o^{D}. This function associates a unique label to each lower-level bag. A *value unshredding* function VUNSHRED performs the opposite conversion, reconstructing the nested collection through an inverse process. High-level definitions of the shredding and unshredding of a value of type T are straightforward, defined by induction on T [20]. Value unshredding can be expressed within the $\mathrm{NRC}^{\mathit{Lbl}+\lambda}$ language. The interesting inductive step is for a tuple type $T = \langle a_1 : T_1 \dots a_n : T_n \rangle$, with $\mathrm{VUNSHRED}_T(o^{\mathrm{F}}, o^{\mathrm{D}}_{\sigma_1} \dots o^{\mathrm{D}}_{\sigma_k}) = \langle a_1 := E_1, \dots a_n := E_n \rangle$. If T_i is a scalar type, then $E_i = o^{\mathrm{F}}.a_i$; otherwise, if T_i is a bag type $\mathit{Bag}(S_i)$, then:

$$\begin{aligned}
E_i = {} & \mathtt{for}\ t\ \mathtt{in}\ \mathtt{Lookup}(o^{\mathtt{Dict}}_{a_i}, o^{\mathrm{F}}.a_i)\ \mathtt{union} \\
& \quad \mathrm{VUNSHRED}_{S_i}(t, o^{\mathrm{D}}_{a_i\sigma'_1} \dots o^{\mathrm{D}}_{a_i\sigma'_j})
\end{aligned}$$

Value shredding has an analogous inductive definition but requires a label-generation function IDGEN(o) that can generate the labels identifying a bag o; for example, IDGEN can return an integer as a unique identifier. Data, or value, shredding is thus a transformation on the values of a nested collection that produces a set of flat collections, collectively referred to as the *shredded data*.

In the recursive view, value dictionaries of type $Label \rightarrow Bag$ are explicitly encoded as a collection of label/bag pairs following the nested data model: $Bag(\langle \texttt{lbl} : Label, \texttt{value} : Bag(T^{\mathrm{F}})\rangle)$. For example, the shredded data of R in recursive view has three components: a top-level bag $\mathtt{R}^{\mathrm{F}}$, and two value dictionaries identified by $\mathtt{R}^{\mathrm{D}}.\mathtt{c}^{\mathtt{fun}}$ and $\mathtt{R}^{\mathrm{D}}.\mathtt{c}^{\mathtt{child}}.\mathtt{e}^{\mathtt{fun}}$.

$$
\begin{aligned}
\mathtt{R}^{\mathrm{F}} &= Bag(\langle\ \mathtt{a}:1,\mathtt{b}:1,\mathtt{c}:1\ \rangle,\langle\ \mathtt{a}:1,\mathtt{b}:2,\mathtt{c}:2\ \rangle,\langle\ \mathtt{a}:2,\mathtt{b}:3,\mathtt{c}:3\ \rangle)\\
\mathtt{R}^{\mathrm{D}}.\mathtt{c}^{\mathtt{fun}} &= Bag(\langle \mathtt{lbl}:1,\mathtt{value}:Bag(\langle\ \mathtt{d}:1,\mathtt{e}:1\ \rangle,\langle \mathtt{d}:2,\mathtt{e}:2\rangle)\rangle,\\
&\qquad \langle \mathtt{lbl}:2,\mathtt{value}:Bag(\langle\ \mathtt{d}:2,\mathtt{e}:1\ \rangle)\rangle,\\
&\qquad \langle \mathtt{lbl}:3,\mathtt{value}:Bag(\langle\ \mathtt{d}:3,\mathtt{e}:3\ \rangle)\rangle)\\
\mathtt{R}^{\mathrm{D}}.\mathtt{c}^{\mathtt{child}}.\mathtt{e}^{\mathtt{fun}} &= Bag(\langle \mathtt{lbl}:1,\mathtt{value}:Bag(\langle\ \mathtt{f}:4\rangle)\rangle,\\
&\qquad \langle \mathtt{lbl}:2,\mathtt{value}:Bag(\langle\ \mathtt{f}:5\ \rangle)\rangle
\end{aligned}
$$

In the relational view, the data shredding of R also has three components: a top-level flat bag $\mathtt{R}^{\mathrm{F}}$, and two value dictionaries $\mathtt{R}^{D}_{\mathtt{c}}$ and $\mathtt{R}^{D}_{\mathtt{c_e}}$.

$$
\begin{aligned}
\mathtt{R}^{\mathrm{F}} &= \{\langle \mathtt{a}:1,\mathtt{b}:1,\mathtt{c}:1\rangle,\langle \mathtt{a}:1,\mathtt{b}:2,\mathtt{c}:2\rangle,\langle \mathtt{a}:2,\mathtt{b}:3,\mathtt{c}:3\rangle\}\\
\mathtt{R}^{D}_{\mathtt{c}} &= \langle\{\langle \mathtt{lbl}:1,\mathtt{d}:1,\mathtt{e}:1\rangle,\langle \mathtt{lbl}:1,\mathtt{d}:2,\mathtt{e}:2\rangle,\langle \mathtt{lbl}:2,\mathtt{d}:2,\mathtt{e}:1\rangle,\langle \mathtt{lbl}:3,\mathtt{d}:3,\mathtt{e}:3\rangle\}\\
\mathtt{R}^{D}_{\mathtt{c_e}} &= \langle\{\langle \mathtt{lbl}:1,\mathtt{f}:4\rangle,\langle \mathtt{lbl}:2,\mathtt{f}:5\rangle\}
\end{aligned}
$$

In the recursive view, value unshredding works bottom-up on the dictionary tree rebuilding the nested object by joining on attributes of *Label*. Once all value dictionaries

are reassociated the result is joined with the top-level to return the final nested collection. In the relational view, value unshredding requires each dictionary to be grouped and then joined. For example, value unshredding on the shredded data of R will first group $\mathtt{R}^{D}_{\mathtt{c_e}}$ by lbl and then join $\mathtt{R}^{D}_{\mathtt{c}}$ on lbl and e. The result is then grouped by lbl and joined with $\mathtt{R}^{\mathrm{F}}$ on c to return the final nested output equivalent to R.

2.5.3 Query Shredding

Once the data is in the shredded representation, the query that operates on the nested values must undergo a query shredding transformation to operate over the shredded data. A query with flat output type is produced for each nested bag attribute in the input query, creating a set of queries collectively referred to as the *shredded query.* Given an input query that operates over nested data to return nested output, the shredded query thus operates over shredded data to return shredded output. Query shredding is a recursive procedure that replaces inner expressions of bag type with labels and produces a corresponding dictionary expression. Thus, a query shredding algorithm takes as input an NRC expression e of type T and produces an expression e^{F} of type T^{F} and a dictionary tree e^{D} of type T^{D}.

This thesis focuses on two previous approaches to query shredding [19, 20]. [20] presented a *symbolic query shredding* approach that produces succinct dictionary expressions built up from λ-terms. The λ-terms provide general recipes for computing a bag expression from an arbitrary label, based on the $\mathit{Label} \to \mathit{Bag}$ dictionary type of the recursive view. Dictionary expressions that are produced from symbolic query shredding are referred to as a *symbolic dictionary expression.* The second strategy referred to as *monolithic shredding* is a path-based translation that produces explicit, *materialized dictionary expressions* that can be evaluated independently [19]. Monolithic shredding works with the relational view and targets SQL directly. Symbolic shredding, monolithic shredding, and the required NRC extensions are now outlined.

Shredded NRC. To support query shredding, NRC is extended with labels, λ-terms, and dictionaries, denoted $\mathrm{NRC}^{\mathit{Lbl}+\lambda}$. The syntax of $\mathrm{NRC}^{\mathit{Lbl}+\lambda}$ is provided in Figure 2.2, extending the previously defined NRC. The semantics of $\mathrm{NRC}^{\mathit{Lbl}+\lambda}$ are provided in Figure 2.3. The λ abstraction is restricted to label parameters: if e is an expression of type T and l is a label variable, then $\lambda l.e$ of type $\mathit{Label} \to T$ can be

$$\begin{aligned} e ::= &\ \textit{[Following NRC Section 2.2]} \\ &\mid \texttt{Label}(\mathit{var}, \ldots) \mid \texttt{match}\ e = \texttt{Label}(\mathit{var}, \ldots)\ \texttt{then}\ e \\ &\mid \texttt{Lookup}(e, e) \mid \lambda \mathit{var}.e \\ &\mid e\ \texttt{DictTreeUnion}\ e \\ &\mid \langle \ldots, a_i : \langle a_i^{\texttt{fun}} : e, a_i^{\texttt{child}} : e \rangle, \ldots \rangle \end{aligned}$$

Figure 2.2: Syntax of $\mathrm{NRC}^{Lbl+\lambda}$.

$$\begin{aligned} [\![\texttt{Lookup}(e_1, e_2)]\!]_p &= [\![e_1]\!]_p([\![e_2]\!]_p) \\ [\![\lambda \mathit{var}.e]\!]_p &= \lambda \mathit{var}.([\![e]\!]_{p[\mathit{var}:=\mathit{var}]}) \\ [\![\texttt{Label}_\tau(\mathit{var}, \ldots)]\!]_p &= \texttt{Label}_\tau(p(\mathit{var}), \ldots) \\ [\![\texttt{match}\ e_1 = \texttt{Label}(\mathit{var}, \ldots)\ \texttt{then}\ e_2]\!]_p &= \begin{cases} [\![e_2]\!]_{p[\mathit{var}:=\mathit{var}\ldots]}\ \mathit{if}\ [\![e_1 = \texttt{Label}(\mathit{var}, \ldots)]\!]_p = \mathit{true} \\ \{\}\ \mathit{if}\ [\![e_1 = \texttt{Label}(\mathit{var}, \ldots)]\!]_p = \mathit{false} \end{cases} \\ [\![e_1 \texttt{DictTreeUnion} e_2]\!]_p &= [\![e_1]\!]_p\ \texttt{DictTreeUnion}\ [\![e_2]\!]_p \\ [\![\langle \ldots, a_i : \langle a_i^{\texttt{fun}} : e_1, a_i^{\texttt{child}} : e_2 \rangle, \ldots \rangle]\!]_p &= \langle \ldots, a_i : \langle a_i^{\texttt{fun}} : [\![e_1]\!]_p, a_i^{\texttt{child}} : [\![e_2]\!]_p \rangle, \ldots \rangle \end{aligned}$$

Figure 2.3: Semantics of $\mathrm{NRC}^{Lbl+\lambda}$.

formed. Similarly, if e_1 is an expression of type $\mathit{Label} \to T$ and e_2 an expression of type Label, then $\texttt{Lookup}(e_1, e_2)$ of type T can be formed; this is standard function application. Label expressions encapsulate variable bindings and include a tag τ that ensures proper application of a label to the corresponding dictionary expression. The $\texttt{Label}(x_1, \ldots, x_n)$ construct creates a new label encapsulating the values of variables $x_1, \ldots, x_n$ of flat types. A label matching construct `match` is provided to deconstruct labels and bind the encapsulated variables in the environment p.

Symbolic shredding algorithm. The symbolic shredding algorithm adapted from [20] takes as input an NRC expression and returns an $\mathrm{NRC}^{Lbl+\lambda}$ expression. Figure 2.4 shows the shredding transformation in terms of recursive functions $\mathcal{F}$ and $\mathcal{D}$. Given a source expression e, the invocation $\mathcal{F}(e)$ returns the expression e^{F} computing the flat version of the output, while $\mathcal{D}(e)$ returns the dictionary tree e^{D} corresponding to e^{F}. The algorithm guarantees that if an expression E has free variables $\mathcal{X}$, then

	Pattern of e	$\mathcal{F}(e)$	$\mathcal{D}(e)$
1	c	c	$\langle\rangle$
2	var	var^{F}	var^{D}
3	$\{e\}$	$\{\mathcal{F}(e)\}$	$\mathcal{D}(e)$
4	$\ldots a_j := e_j, \ldots$ e_j of scalar type$\rangle$	$\ldots a_j := \mathcal{F}(e_j), \ldots\rangle$	$\ldots\rangle$ // omit empty dictionary tree for a_j
5	$\langle\ldots a_i := e_i, \ldots$ e_i of bag type	$\langle\ldots a_i :=$ `Label`$(x^{\mathrm{F}}), \ldots\rangle$	$\langle\ldots a_i^{\texttt{fun}} := \lambda l.$ `match` $l =$ `Label`$(\ldots)$ `then` $\mathcal{F}(e_i)$, $a_i^{\texttt{child}} := \{\mathcal{D}(e_i)\}, \ldots$
6	$e.a$ of bag type	`Lookup`$(\mathcal{D}(e).a^{\texttt{fun}}, \mathcal{F}(e).a)$	`get`$(\mathcal{D}(e).a^{\texttt{child}})$
7	$e.a$ of scalar type	$\mathcal{F}(e).a$	$\langle\rangle$
8	`for` var `in` e_1 `union` e_2	`let` $var^{\mathrm{D}} := \mathcal{D}(e_1)$ `in` `for` var^{F} `in` $\mathcal{F}(e_1)$ `union` $\mathcal{F}(e_2)$	`let` $var^{\mathrm{D}} := \mathcal{D}(e_1)$ `in` $\mathcal{D}(e_2)$
9	`let` $var :=$ e_1 `in` e_2	`let` $var^{\mathrm{D}} := \mathcal{D}(e_1)$ `in` `let` var^{F} `in` $\mathcal{F}(e_1)$ `in` $\mathcal{F}(e_2)$	`let` $var^{\mathrm{D}} := \mathcal{D}(e_1)$ `in` $\mathcal{D}(e_2)$
10	`if` $cond$ `then` e	`if` $\mathcal{F}(cond)$ `then` $\mathcal{F}(e)$	$\mathcal{D}(e)$
11	$e_1 \uplus e_2$	$\mathcal{F}(e_1) \uplus \mathcal{F}(e_2)$	$\mathcal{D}(e_1)$ `DictTreeUnion` $\mathcal{D}(e_2)$
12	$op(e_1, e_2)$	$op(\mathcal{F}(e_1), \mathcal{F}(e_2))$	$\langle\rangle$

Figure 2.4: Query shredding algorithm for NRC

the output E^{F} has free variables contained in the *shredded variables* corresponding to $\mathcal{X}$. For a variable x of type T, the two functions return shredded variables x^{F} and x^{D} whose types T^{F} and T^{D} depend on T as described above (line 2). For a scalar c, this gives $\mathcal{F}(c) = c$ and $\mathcal{D}(c)$ is an empty dictionary tree (line 1).

The interesting cases are tuple construction and projection. The tuple constructor (line 4-5) builds a recursive dictionary tree with $a_i^{\texttt{fun}}$ representing the mapping from a label to the flat variant e_i^{F} of e_i and $a_i^{\texttt{child}}$ representing the dictionary tree for e_i^{F}. To conform with the type system, each child dictionary tree is wrapped as a singleton bag as seen in Section 2.5.1. For each scalar-valued attribute a_j in a tuple constructor (line 4), $\mathcal{F}$ recurs on the scalar expression e_j to produce e_j^{F}. Since e_j is already flat, e_j^{D} is empty it is omitted from the dictionary tree of the tuple constructor. For each bag-valued attribute a_i in a tuple constructor (line 5), $\mathcal{F}$ replaces the bag expression e_i with a new label encapsulating the (flat) free variables of e_i. When projecting a bag-valued attribute a on a tuple expression e (line 6), $\mathcal{F}$ returns a `Lookup` construct computing the flat bag of $e.a$, based on the dictionary $\mathcal{D}(e).a^{\texttt{fun}}$ and label $\mathcal{F}(e).a$ formed during tuple construction. The returned child dictionary tree serves to dereference any

label-valued attributes in the corresponding flat bag. When $e.a$ is a scalar expression (line 7), $\mathcal{F}$ recursively computes $e^{\mathrm{F}}.a$, while $\mathcal{D}$ returns an empty dictionary tree.

For binary union, the same label-valued attribute may correspond to labels that depend on two different sets of free variables. To encapsulate two dictionary trees, the function $\mathcal{D}$ constructs a `DictTreeUnion` of tuple type (line 11). For the remaining constructs, the functions $\mathcal{F}$ and $\mathcal{D}$ proceed recursively, maintaining the invariant that for any source expression e, the free variables of e^{F} or e^{D} in the shredded representation correspond to the free variables of e.

An example of the symbolic query shredding algorithm is provided on `Ex2`, producing top-level expression $\mathtt{Ex2}^{\mathrm{F}}$ and symbolic dictionary expression $\mathtt{Ex2}^{\mathrm{D}}$. `Ex2` is first matched to the `for` construct in $\mathcal{F}$ and the process recurs to return $\mathtt{Ex2}^{\mathrm{F}}$ with a $\mathtt{Label}_{\mathtt{g}}$ expression in replace of `g` that encapsulates $\mathtt{r}^{\mathrm{F}}$ since it is a free variable in the subexpression at `g`. `Ex2D` is then derived matching the `for` construct in $\mathcal{D}$ and recurring. The bag type expression at `g` produces an $a_i^{\mathtt{fun}}$ with $\mathtt{r}^{\mathrm{F}}$ in scope from matching incoming labels of $\mathtt{Label}_{\mathtt{g}}$. The resulting shredded query of `Ex2` is:

$$
\begin{array}{l}
\mathtt{Ex2}^{\mathrm{F}} \Leftarrow \\
\quad \mathtt{for}\ \mathtt{r}^{\mathrm{F}}\ \mathtt{in}\ \mathtt{R}^{\mathrm{F}}\ \mathtt{union} \\
\qquad \{\langle \mathtt{a} := \mathtt{r}^{\mathrm{F}}.\mathtt{a}, \mathtt{g} := \mathtt{Label}_{\mathtt{g}}(\mathtt{r}^{\mathrm{F}})\rangle\}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{Ex2}^{\mathrm{D}} \Leftarrow \\
\quad \mathtt{let}\ \mathtt{r}^{\mathrm{D}} := \mathtt{R}^{\mathrm{D}}\ \mathtt{in} \\
\quad \langle \mathtt{g}^{\mathtt{fun}} : \lambda l.\mathtt{match}\ l = \mathtt{Label}_{\mathtt{g}}(\mathtt{r}^{\mathrm{F}}) \\
\qquad \mathtt{for}\ \mathtt{c}^{\mathrm{F}}\ \mathtt{in}\ \mathtt{Lookup}(\mathtt{r}^{\mathrm{D}}.\mathtt{c}^{\mathtt{fun}}, \mathtt{r}^{\mathrm{F}}.\mathtt{g})\ \mathtt{union} \\
\qquad\quad \mathtt{for}\ \mathtt{s}^{\mathrm{F}}\ \mathtt{in}\ \mathtt{S}^{\mathrm{F}}\ \mathtt{union} \\
\qquad\qquad \mathtt{if}\ \mathtt{c}^{\mathrm{F}}.\mathtt{d}\ ==\ \mathtt{s}^{\mathrm{F}}.\mathtt{a}\ \mathtt{then} \\
\qquad\qquad\quad \{\langle \mathtt{d} := \mathtt{c}^{\mathrm{F}}.\mathtt{d},\ \ \mathtt{b} := \mathtt{r}^{\mathrm{F}}.\mathtt{b} * \mathtt{s}^{\mathrm{F}}.\mathtt{b}\rangle\}), \\
\qquad \mathtt{g}^{\mathtt{child}} : \langle\rangle\rangle
\end{array}
$$

While the symbolic dictionary expression $\mathtt{Ex2}^{\mathrm{D}}$ is succinct and encodes the nested structure of `Ex2`, the shredded query that results from symbolic query shredding must undergo an additional transformation to materialize dictionaries that are explicit encodings of label/bag pairs and free of λ-terms. The process of materializing explicit dictionary expressions from a symbolic dictionary expression is referred to as *materialization*; however, [20] does not provide an implementation and thus did not target explicit materialized dictionary expressions.

Monolithic Shredding. In contrast to symbolic shredding, *monolithic shredding* results in materialized dictionary expressions that are free of λ expressions and are in a form suitable for evaluation. Monolithic shredding generates stand-alone subqueries in SQL [19] that are well-suited for the parallel execution strategy of relational query engines. Collectively, these subqueries make up the shredded query, with one query component for the top-level bag and one component for each lower-level dictionary. Since each query is designed to be executed independently, however, the component query for a dictionary D will need to independently materialize the label domain for D. The independent reconstruction of labels in each subquery can result in duplicated computation of different components as well as repetitive iteration steps, which could be problematic for queries with several levels of nesting.

The monolithic shredding of the `Ex2` NRC expression produces a top-level expression $\texttt{Ex2}^{\mathrm{F}}$ and a materialized dictionary identified by path $\texttt{Ex2}^{\mathrm{D}}_{\mathrm{g}}$:

$$
\begin{array}{l}
\texttt{Ex2}^{\mathrm{F}} \Leftarrow \\
\quad \texttt{for } \mathtt{r}^{\mathrm{F}} \texttt{ in } \mathtt{R}^{\mathrm{F}} \texttt{ union} \\
\qquad \{\langle \mathtt{a} := \mathtt{r}^{\mathrm{F}}.\mathtt{a}, \mathtt{g} := \mathtt{Label}_{\mathtt{g}}(\mathtt{r}^{\mathrm{F}}) \rangle\}
\end{array}
$$

$$
\begin{array}{l}
\texttt{Ex2}^{\mathrm{D}}_{\mathtt{g}} \Leftarrow \\
\quad \texttt{for } \mathtt{r}^{\mathrm{F}} \texttt{ in } \mathtt{R}^{\mathrm{F}} \texttt{ union} \\
\qquad \texttt{for } \mathtt{c}^{\mathrm{F}} \texttt{ in } \mathtt{Lookup}(\mathtt{r}^{\mathrm{D}}.\mathtt{c}, \mathtt{r}^{\mathrm{F}}.\mathtt{g}) \texttt{ union} \\
\qquad\quad \texttt{for } \mathtt{s}^{\mathrm{F}} \texttt{ in } \mathtt{S}^{\mathrm{F}} \texttt{ union} \\
\qquad\qquad \texttt{if } \mathtt{c}^{\mathrm{F}}.\mathtt{d} \ == \ \mathtt{s}^{\mathrm{F}}.\mathtt{a} \texttt{ then} \\
\qquad\qquad\quad \{\langle \mathtt{lbl} := \mathtt{Label}_{\mathtt{g}}(\mathtt{r}^{\mathrm{F}}), \mathtt{d} := \mathtt{c}^{\mathrm{F}}.\mathtt{d}, \ \mathtt{b} := \mathtt{r}^{\mathrm{F}}.\mathtt{b} * \mathtt{s}^{\mathrm{F}}.\mathtt{b} \rangle\}
\end{array}
$$

Since [19] targeted SQL directly, the shredded query above is only a manual translation of NRC to $\mathrm{NRC}^{Lbl+\lambda}$ following monolithic shredding. Note that the materialized dictionary expression contains no λ expressions and follows the relational view.

Our Approach. While the shredding techniques discussed in this section provide a foundation for adapting shredding to distributed processing frameworks, they fall short in a few areas. First, the symbolic shredding transformation does not produce explicit expressions that can be compiled to a distributed application. Second, monolithic shredding produces explicit expressions designed for a centralized setting that executes queries in parallel independently, rather than the sequential execution of distributed processing platforms. This thesis takes a two-stage query shredding approach that first performs symbolic query shredding that adapts [20] to NRC with aggre-

gation. The second stage applies a novel materialization strategy to eliminate λ abstractions and produce an explicit representation of the shredded query designed for sequential execution on a distributed framework.

2.6 Distributed Processing Frameworks

Large-scale data analysis platforms were designed to automatically handle the complexities of distributed computing and parallelization, and have become indispensable tools for modern data analysis. MapReduce [3] was the earliest model for distributed programming environments, which was implemented in Hadoop [38]. The MapReduce model consists of a map phase that performs sorting or filtering operations, and then a reduce stage such as aggregation or counting. The downside of MapReduce was that results between map and reduce stages had to be materialized on disk. Inspired by but not strictly following the MapReduce model, Apache Spark [1] and Apache Flink [2] were developed to support caching of data into memory and minimizing writes of intermediate data to physical storage. This section extends the discussion of distributed processing systems from Chapter 1 using Spark as the representative system.

The core of Spark uses a specialized data structure for representing distributed data, known as a Resilient Distributed Dataset (RDD) [39]. An RDD is a collection of distributed objects, where a *partition* is the smallest unit of distribution. Partitions are local collections that only exist on a single node. For example, if the cluster consists of two nodes, then there may be five partitions on one node and five partitions on the other node. All ten partitions make up an RDD. When a flat data source, such as `Part`, is imported into Spark each item of the collection is allocated in round-robin fashion to each partition. The number of partitions can be user-specified or automatically estimated based on input size. The same, top-level distribution strategy is applied when importing a nested dataset.

Figure 1.1 shows the setup of a Spark cluster. An application, defined in the distributed collection API, is submitted to the coordinator node, which then delegates tasks to worker nodes in a highly distributed, parallel fashion. The distributed collection API communicates high-level analytical tasks to the coordinator while abstracting data distribution and task delegation from the user; thus, a user never communicates

```
1  case class FlatCOP(cname: String, odate: String, pid: Int, qty: Double)
2  case class PartType(pid: Int, pname: String, price: Double)
3
4  val COP = spark.read.json("cop.json").as[COPType]
5  val Part = spark.json.csv("part.csv").as[PartType]
6
7  val result = COP.flatMap(cop =>
8               cop.corders.flatMap(co =>
9                 co.oparts.flatMap(op =>
10                  FlatCOP(cop.cname, co.odate, co.pid, co.qty))))
11              .join(Part, "pid")
12              .groupByKey(s => ("cname", "odate"))
13
14 result.count
```

Figure 2.5: Example SparkSQL application that uses the Dataset API to join flattened `COP` with `Part` and then group by customer name and order date.

with a worker node directly. Tasks are submitted as batch operations over distributed collections that are executed sequentially, such that an operation is complete when all tasks are complete and the runtime is dependent on the slowest running task.

SparkSQL. SparkSQL [40] provides a relational framework on top of the core of Spark, which uses a strongly-typed, immutable collection of objects known as an *Dataset*. Datasets map to relational schemas and represent lazy query expressions that produce logical query plans with Sparks query planner, Catalyst [40]. After a series of execution stages to optimize the resulting query plan, a Dataset is compiled to a graph of RDD operations [41]; thus, Datasets are relational abstractions of RDDs that provide additional opportunities for optimization. In addition, Datasets use encoder objects to significantly reduce the memory footprint and serialization costs of RDDs.

Operations over Datasets can be defined directly in SQL or using a specialized collection API, known as the Dataset API. An example distributed application using the Spark/Scala Dataset API is presented in Figure 2.5. The program associates nested parts from `COP` to the relevant parts in `Part`; this is the association required for the `Totals` example. The result is then regrouped by `cname` and `odate`. Figure 2.6 presents an identical SparkSQL program using the SQL interface instead of the Dataset API; both programs follow identical paths to execution.

The program starts by defining a case class, named `FlatCOP` (line 1), which encapsu-

```
spark.sql("""
  SELECT cop.cname, co.odate, collect_list(named_struct("pid",
      op.pid, "qty", op.qty, "pname", p.pname, "price", p.price))
  FROM (
    SELECT cop.cname, co.odate, op.pid, op.qty, p.pname, p.price
        FROM COP as cop
    LATERAL VIEW EXPLODE(cop.corders) as co
    LATERAL VIEW EXPLODE(co.oparts) AS op
    JOIN Part as p ON op.pid = p.pid
    GROUP BY cop.cname, co.odate
  """)
```

Figure 2.6: Equivalent SparkSQL program from Figure 2.5 defined using SQL instead of the Dataset API.

lates objects of type $Bag\,(\langle\,\texttt{cname} : \mathit{string}, \texttt{odate} : \mathit{string}, \texttt{pid} : \mathit{int}, \texttt{qty} : \mathit{real}\rangle)$. Case classes are light-weight classes provided natively in Scala. For this example, `COPType` is the case class representation for the `COP` data type and `PartType` is the case class representation for the `Part` data type. The program then reads the inputs, `COP` and `Part`, from JSON and CSV files on disk (lines 3-4) to produce Datasets. The `flatMap` operation (lines 6-9) works locally at each partition, iterating over top-level objects in `COP`, navigating into `corders`, and then into `oparts` to create instances of `FlatCOP` objects. The `join` operator (line 10) merges tuples from the result of `flatMap` and `Part` based on the equality of `pid`, which is the key of the key-based partitioning guarantee that shuffles matching values to the same partition. The final `groupByKey` operation (line 11) groups the joined result based on unique (`cname`, `odate`) values, and sends all tuples with matching key values (`cname`, `odate`) to the same partition. The program defines a collection of tuples with final output type `result`:$Bag\,(\langle\,\texttt{cname} : \mathit{string}, \texttt{odate} : \mathit{string}, _2 : Bag\,(\langle\,\texttt{pid} : \mathit{int}, \texttt{qty} : \mathit{real}, \texttt{pname} : \mathit{string}, \texttt{price} : \mathit{real}\rangle)\,\rangle)$.

When `count` (line 13) is called, `result` is transformed into an optimized execution plan. The Catalyst optimizer determines the optimal plan based on standard database optimizations, such as join order, join strategy, and projection pushing. For this example, the optimizer chooses to use a Sort Merge Join for the execution associated to line 10. Both the `FlatCOP` and `Part` Datasets are sorted based on `pid` and the join is performed based on a hash-partitioning strategy. The hash-partitioner uses the hash of the `pid` to send `FlatCOP` and `Part` tuples with the same `pid` values to the same parti-

tion, following the key-based partitioning strategy.

Spark programmers will often optimize their programs by minimizing the amount of shuffled data. For instance, after the `groupByKey` operation the Dataset associated to `result` will be partitioned by (`cname`, `odate`). Subsequent operations could exploit this partitioning strategy to minimize the amount of shuffled data.

High-Level Languages. Users can also interact with the system through higher-level, declarative languages that provide additional optimizations and reduce the amount of hand-tuning required for an application. Some high-level solutions provide scripting languages, such as Apache Pig [42] and Scope [12], and others provide SQL interfaces, such as HIVE [13] and SparkSQL [40]. The optimizers of these languages are still in early stages of development, so the plan generated from the above SparkSQL programs may not be as efficient as a hand-tuned program from an experienced Spark developer. Further, the core APIs and higher-level languages of these frameworks are are unable to optimize analytical tasks that operate on nested collections. For example, Apache Pig uses an explicit flattening operation that will result in a cross product of the input table; this is exactly the plan that results from the unnest operator (Section 2.4). Optimizations are provided in Pig that can reduce the amount produced by flattening, ie. "PushDownForEachFlatten", but this is only useful for selective queries. At the core of it all, Apache Pig, SparkSQL, and similar approaches all use flattening techniques that lead to performance issues when processing nested data.

Our Approach. The framework presented in this thesis compiles directly to SparkSQL via the Dataset API to leverage all optimization opportunities from the internal Spark optimizer. The compilation also supports compiling shredded queries directly to SparkSQL which overcomes the limitations of flattening and takes full advantage of the Spark optimizer without the burden of the nested representation.

2.7 Skew-Handling Techniques

This section discusses skew-handling techniques specific to distributed processing platforms. Skew-handling involves the proper management of *heavy keys*. A key is considered heavy when that key has many more associated values than other keys. The partition housing a heavy key will produce runtime bottlenecks in distributed execution.

Further, the partition could be completely saturated leading to the burden of reading and writing to disk as noted above.

Skew-handling includes both the identification of heavy keys and the management of values associated to the the heavy keys [21]. There are two approaches for skew-handling - redistribution and duplication. Redistribution is a strategy in which heavy keys are randomized and a hash-based partitioning strategy is used to ensure the distribution of the values associated to heavy keys. Load imbalances can arise with redistribution since randomization does not guarantee a balanced partitioning strategy; thus, this is dependent on an optimal, data-specific hash function. One approach is to construct the hash from local and global histograms [22]. Local histograms are built for each partition on the input using only the key information, and are then consolidated globally to identify heavy keys. The global histogram and redistribution strategy for the heavy keys are based on a pre-join of the data which can be expensive. Redistribution also requires moving all values, which can lead to high network costs.

The duplication strategy will *broadcast* one relation to all other nodes and perform the join locally. Broadcast is a feature of distributed processing platforms that copies a collection to each node in the cluster in order for operations to access it locally. While this avoids the movement of all tuples, there are overheads when the broadcasted relation is large. To solve this, input relations can be split based on heavy and light keys and only the tuples with heavy keys from one relation are broadcast. Tuples associated with light keys are handled with a standard join and the result of the join is the union of both [23]. This is referred to as Partial Redistribution and Partial Duplication (PRPD) since light values are redistributed and duplication techniques are used for heavy values. PRPD is one of the strongest approaches for inner-joins, but their method assumes that heavy keys are known. The skew-resilience techniques in this thesis build upon the PRPD approach, expanding the skew-handling procedure to generate light and heavy query plans that adapt based on the identification of heavy keys at runtime.

Skew-handling research is primarily focused on the implementation of inner, equi-joins implementations. Research on sampling has been explored in the context of building histograms for redistribution [22, 43]. To the best of our knowledge, the practical implementation of sampling techniques for PRPD techniques has not been explored. Though skew can be worse when dealing with nested data, skew-handling has not been

specifically explored in the context of nested data processing. Similarly, skew-handling in the context of shredding has not been explored even though shredded representations can minimize the skew that arises from large or varying-sized inner collections. The thesis extends skew-resilient query processing techniques, focusing on the use of sampling techniques in adaptable plans for both nested and shredded representations.

2.8 Implementation of Nested Query languages

The chapter thus far has discussed the nested data model, associated languages, and potential optimizations. Some key systems that leverage these techniques to operate on nested collections are presented next.

2.8.1 Query Language Implementations

Query language implementations for complex object processing can be stand-alone or extensions to existing frameworks. The object relational approaches were supported by SQL3 and the SQL:1999 standard, which extended support for relational DBMSs. For object oriented databases, [10] translated OQL to monoid comprehensions, which extended the object data model to provide optimization support to OODBs. Kleisli, the NRC-based system mentioned above, was motivated by the biological analysis and data access mechanisms of the Human Genome Project [37, 44]. These biological data sources did not follow the relational model and were not uniformly accessible through any existing solution. Kleisli was implemented as a stand-alone compilation framework that worked with the nested model to support these complex datatypes.

Other systems use language-integrated approaches. The Links [45] and Linq [46] systems compile their high-level queries into XML and SQL to overcome the impedance mismatch. XML, like JSON, is a semi-structured data type that is richer than the nested model - supporting heterogeneous types, missing attributes, and ordering. There are recent document stores that are provided as DMBSs with non-standardized languages and target XML and JSON [6, 4, 5].

2.8.2 Internal Data-Specific Systems

Flattened internal data representations are leveraged by Google's large scale analytics systems such as Spanner [47], F1 [14, 15], and Dremel [7]. These are stand-alone architectures that provide "Standard SQL" language dialects with proprietary backends. The data model of these systems combines relational and tree structures [48] to process JSON and Protocol Buffers. Protocol Buffers are another data type that is richer than the nested model, supporting missing attributes in a structured schema.

Shredded representations and query shredding have been utilized by in the Links [19] and Ferry [49, 50] systems. Links compiles high-level queries to SQL, supporting nested data through XML. The Ferry system implements shredding that is specific to SQL translations from XQuery, a functional query language that works on XML. Both of these systems focus on the generation of SQL queries and target parallel execution on relational query engines. An additional shredding algorithm is proposed that focuses on reducing incremental evaluation of nested queries to incremental evaluation of flat queries [20]. This is merely a transformation, with no associated implementation, that focuses on dynamic evaluation of an NRC variant without aggregation.

2.8.3 Distributed Processing-Specific Systems

Some systems provide distributed processing capabilities for large-scale datasets. Backend data storage can physically be distributed across worker nodes, which is the approach taken by relational engines and document stores. Citus [51] is a relational engine with collection support that executes queries over Postgres instances distributed over several nodes. MongoDB [6] is a document store that can use a central node to communicate with distributed nodes that operate on chunks of data, known as *shards*. Both of these systems abstract distribution from the user, but enforce limitations on queries that operate over distributed data sources. Only one of the sources involved in a join can be sharded in MongoDB. Citus does not directly support joins between relations partitioned on different columns. A distributed view can be materialized and partitioned on a new join key, but for nested data this means materializing the entire flattened object. The backend storage in these systems must be manually configured on each node. More details on these systems can be found in Chapter 7.

Modern distributed processing frameworks automate data distribution and support distributed operations with less limitations. Several existing frameworks for processing nested data target these frameworks. Emma [16] handles programming abstractions by deeply embedding their source language into a host language, like Scala. DIQL [52] and MRQL [53] also provide embedded DSLs in Scala that perform normalization and decorrelation to generate efficient Spark and Flink programs. This work extends the monoid algebra from [10] and thus extends the previously mentioned, ODMG model to support comprehensions with grouping, ordering, and cogrouping operations. Rumble [54] operates on the nested representation by translating JSONiq [55], a query language for JSON, to Spark. This framework is specifically designed to deal with the challenges associated with richer nested data models. All the systems that target distributed frameworks employ flattening techniques to support nested data processing, despite the known limitations of the standard nested representation first discussed in Chapter 1.

2.9 User-Defined Functions

To provide extensibility, most query languages allow users to extend the language with their own transformations called UDFs. The complexity of data science pipelines often means that declarative queries will leverage functions that perform tasks such as classification or clustering [56] since this is not tractable in standard source languages.

To support such advanced operations, declarative languages can provide UDFs that are treated as *black-box* operators alongside standard relational operators. Where black-box describes the unrecognizable nature of a UDF to a query optimizer, both in the internal operations invoked and cost estimations. Earlier work for the optimization of UDFs in relational query engines relied on user-provided annotations, such as estimated expense [57] and binding access patterns [58], that can help direct an optimizer to make better decisions and reduce the space of query plans.

User-provided annotations, sometimes known as hints, avoid the need for complex program analysis and are thus a common approach for UDF optimization. Hints have been used to optimize UDFs leveraged by distributed processing applications, where the focus is on reducing the amount of shuffling that could result from executing a

UDF [56, 59]. None of these approaches have considered user-provided hints that are agnostic to distribution, but can still leverage aspects of distribution in their implementation. Further, user-defined functions have not been specifically explored in the context of nested data processing or in combination with shredding techniques. This thesis explores the use of user-defined functions with shredded representations.

2.10 Thesis Motivation

It is clear that current approaches to processing nested collections in distributed settings will lead to performance issues. While some approaches have handled the programming mismatch, none have explored query compilation, shredding, and skew-resilience within the same context. Further, existing shredding techniques are either incomplete for compiling to distributed programs or are designed around a centralized setting. This chapter has presented the basis of these techniques in the context of our framework and source NRC language. The rest of this thesis presents the extensions of the compilation framework to support aggregation, a novel shredding transformation for targeting distributed processing, and associated skew-resilient processing.

Chapter 3

Compiling NRC with Aggregation

This chapter presents the extensions to NRC and compilation components to support aggregation and compile to a distributed application. The aggregation extensions are collectively referred to as NRC_{agg} and are presented in Section 3.1. The extensions to the plan language to support aggregation are then presented in Section 3.2, followed by the relevant plan translation rules in Section 3.3. The code generation process that produces a distributed application from a plan is described in Section 3.4. The components of this section collectively compile a query over nested data into a distributed application that operates on the nested representation; this is referred to as the *standard compilation route*.

3.1 NRC with Aggregation

This work provides a source language, NRC_{agg}, which extends the previously-defined language NRC (Section 2.2) with key-based operations commonly used in distributed collection APIs. NRC_{agg} works with the nested data model, with no explicit support for set types since these can be modeled as bags with multiplicity one. A bag of tuples where each attribute has a scalar type is referred to as a *flat bag*. Figure 3.1 gives the syntax of NRC_{agg}. For completeness, the standard NRC expressions are provided

$$
\begin{aligned}
cond ::= &\ e\ RelOp\ e \mid \neg cond \mid cond\ BoolOp\ cond \\
e ::= &\ \emptyset_{Bag(F)} \mid \{e\} \\
&\mid var \mid e.a \mid \langle a_1 := e, \ldots, a_n := e \rangle \\
&\mid \texttt{for}\ var\ \texttt{in}\ e\ \texttt{union}\ e \mid e \uplus e \\
&\mid \texttt{if}\ cond\ \texttt{then}\ e\ \texttt{else}\ e \\
&\mid \texttt{let}\ var := e\ \texttt{in}\ e \mid e\ PrimOp\ e \\
&\boxed{\begin{aligned} &\mid \texttt{groupBy}_{key}^{value}(e) \mid \texttt{sumBy}_{key}^{value}(e) \\ &\mid \texttt{get}(e) \mid \texttt{dedup}(e) \mid \texttt{udf}_{params}^{name}(e) \end{aligned}}
\end{aligned}
$$

Figure 3.1: Syntax of NRC_{agg}.

and the extensions specific to this thesis are outlined. User facing NRC_{agg} does not support comprehension-syntax; comprehension-syntax is only supported internally to facilitate downstream optimizations.

NRC_{agg} programs are sequences of assignments ($\Leftarrow$) of variables to expressions. The `get`(e) operation takes a singleton bag and returns its only element; if e is an empty bag or a bag with more than one element, `get` returns a default value. The language provides deduplication and key-based operations for grouping and sum aggregation. Since the language does not support set types, these operations are *multiplicity-altering* meaning all multiplicities of the output bag are set equal to one. The deduplication operation `dedup`(e) takes the bag e and returns a bag with the same elements, but with all multiplicities changed to one. The input to `dedup` is restricted to be a flat bag.

The `groupBy` and `sumBy` expressions perform key-based grouping and aggregation, respectively. The *key* attributes for these functions are required to be flat. The grouping operator $\texttt{groupBy}_{key}^{value}(e)$ groups the tuples of bag e by the collection of *key* attributes. For each distinct *key* value, a bag is produced containing the tuples from e with only the *value* attributes projected. The user can specify the named attribute for the grouped bag, which defaults to _2.

The sum aggregate operator $\texttt{sumBy}_{key}^{value}(e)$ groups the tuples of bag e by each distinct

$$
\begin{aligned}
[\![\texttt{get}(\{e\})]\!]_p &= [\![e]\!]_p \\
[\![\texttt{dedup}(e)]\!]_p &= \bigcup{}_{v \in [\![e]\!]_p} p(v) \\
[\![\texttt{groupBy}^{value}_{key}(e)]\!]_p &= \bigcup{}_{k \in [\![\pi_{key}(e)]\!]_p} [\![\langle k, \biguplus{}_{v \in [\![e]\!]_p} [\![\pi_{value}(\sigma_{k=\pi_{key}(v)}(e))]\!]_p \rangle]\!]_p \\
[\![\texttt{sumBy}^{value}_{key}(e)]\!]_p &= \bigcup{}_{k \in [\![\pi_{key}(e)]\!]_p} [\![\langle k, \biguplus{}_{v \in [\![e]\!]_p} \texttt{sum}([\![\pi_{value}(\sigma_{k=\pi_{key}(v)}(e))]\!]_p) \rangle]\!]_p \\
[\![\texttt{udf}^{name}_{params}(e)]\!]_p &= \texttt{udf}^{name}_{params}([\![e]\!]_p)
\end{aligned}
$$

Figure 3.2: Semantics of the NRC_{agg} extensions.

key value and returns the sum of each attribute in *value*. The `sumBy` operator will perform implicit projection if any of the attributes from e are not in the *key* or *value* set; these are referred to as *key-value* tuples. An example of a key-value tuple type is presented in the next section.

To support more complex aggregation and analytical tasks, the source language provides a user-defined function construct that supports both native and external function calls. The UDF operation $\texttt{udf}^{name}_{params}(e)$ is identified by the *name* attribute and takes an NRC_{agg} expression as input. The syntax supports additional parameters specific to the UDF. More details on UDF compilation are provided in Section 6.3.

3.1.1 Semantics

Figure 3.2 presents the semantics of NRC_{agg}, extending the semantics from Section 2.2. As previously, let p range over environments mapping variables to values, with obvious bindings $p[k \to k]$ omitted. In addition to the previously used bag union $\biguplus$, set union $\bigcup$ is used to express the removal of duplicates (`dedup`). The semantics for `sumBy` and `groupBy` make use of standard relational operations, π and σ. Projection π is used to denote the dropping of fields from each tuple in the collection, ie. for `S` (Section 2.1) $\pi_a(\texttt{S})$ returns $\{\langle\ a\text{: } 1\ \rangle, \langle\ a\text{: } 2\ \rangle\}$. Selection is used to reflect filtering on a relation, ie. $\sigma_{a=1}(\texttt{S})$ returns $\{\langle\ a\text{: } 1,\ b\text{: } 5\ \rangle\}$. The `sumBy` operation uses an additional function `sum` that given an input collection returns the sum across each attribute. In the example above, `sum(S)` returns the tuple $\langle a : 3, b : 11\rangle$.

$$
\begin{array}{lcl}
\Gamma^{\uplus_{gb}/value}_{key} X & = & \texttt{groupBy}^{value}_{key}(X) \\
\Gamma^{+/value}_{key} X & = & \texttt{sumBy}^{value}_{key}(X) \\
\texttt{dedup} X & = & \texttt{groupBy}^{\emptyset}_{\overline{a}}(X) \\
\texttt{udf}^{name}_{params} X & = & \texttt{udf}^{name}_{params}(X)
\end{array}
$$

Figure 3.3: Aggregate plan language operators and their semantics.

3.2 Plan Language Extensions

The intermediate object algebra presented in Section 2.3 is extended to support the key-based aggregation from NRC_{agg}. The aggregation operators of the plan language and their semantics are presented in Figure 3.3. Following the presentation of the base operators, $\overline{a}$ denotes a sequence of attributes $a_1, \ldots, a_k$ for each tuple x in X. Key-based grouping $\Gamma^{\uplus_{gb}/value}_{key}$ follows the semantics of `groupBy`. This is similar to the nest operator $\Gamma^{\uplus/\overline{v}}_{\overline{a}}$ from Section 2.3 where $\overline{a}$ has no duplicates. The key-based summation operator $\Gamma^{+/value}_{key}$ is referred to as the sum aggregate and follows the semantics of `sumBy`. As in the standard plan language, the Γ operator casts `NULL` values introduced by the outer operators to 0 for the $+$ aggregate and the empty bag for the $\uplus$ aggregate. The `dedup` plan operator can be modeled as `groupBy` with an empty value set, thus producing a distinct set of tuples. Any user-defined function will be represented by the associated `udf` operator that is passed through from the NRC_{agg} expression. These plan operators provide an intermediate representation of NRC_{agg} to facilitate optimizations and code generation.

3.3 Plan Translation Extensions

This section discusses the additional rules of plan translation (Section 2.4) to support aggregation. A walk-through of the extended plan translation procedure on the `Totals` example is provided.

Translation starts from the outermost level of a comprehension, applying the plan translation rules (U1 - U17) to recursively build up a subplan E from a comprehension

Key-based sum aggregate plan translation rules		
U9 `sumBy` in the head u is empty, w and E are non-empty		
$\texttt{PTrans}(\{\texttt{sumBy}^v_k(\{e \mid \overline{r}\}) \mid p\})^{()}_w E$	$=$	$\Gamma^{+/v}_k \sigma_p(\texttt{PTrans}(\{e \mid \overline{r}\})^{()}_w E)$
U10 `sumBy` in the head u, w and E are non-empty		
$\texttt{PTrans}(\{\texttt{sumBy}^v_k(\{e \mid \overline{r}\}) \mid p\})^{u}_w E$	$=$	$\Gamma^{\uplus/(k,v)}_u \sigma_p(\texttt{PTrans}(\texttt{sumBy}^v_{k^u}(\{e^u \mid \overline{r}\}))^{()}_w E)$
U11 `sumBy` in the body w and E are non-empty		
$\texttt{PTrans}(\{e_1 \mid \overline{s}, y \leftarrow \texttt{sumBy}^v_k(\{e \mid \overline{r}\})\})^{u}_w E$	$=$	$\texttt{PTrans}(\{e_1 \mid \overline{s}\})^{u}_{\alpha^y_I}(\texttt{PTrans}(\texttt{sumBy}^v_{k^w}\{e^w \mid \overline{r}\})^{()}_w E)$
Key-based grouping plan translation rules		
U12 `groupBy` in the head u is empty, w and E are non-empty		
$\texttt{PTrans}(\{\texttt{groupBy}^v_k(\{e \mid \overline{r}\}) \mid p\})^{()}_w E$	$=$	$\Gamma^{\uplus_{gb}/v}_k \sigma_p(\texttt{PTrans}(\{e \mid \overline{r}\})^{()}_w E)$
U13 `groupBy` in the head u, w and E are non-empty		
$\texttt{PTrans}(\{\texttt{groupBy}^v_k(\{e \mid \overline{r}\}) \mid p\})^{u}_w E$	$=$	$\Gamma^{\uplus/(k,_2)}_u \sigma_p(\texttt{PTrans}(\texttt{groupBy}^v_{k^u}(\{e^u \mid \overline{r}\}))^{()}_w E)$
U14 `groupBy` in the body w and E are non-empty		
$\texttt{PTrans}(\{e_1 \mid \overline{s}, y \leftarrow \texttt{groupBy}^v_k(\{e \mid \overline{r}\})\})^{u}_w E$	$=$	$\texttt{PTrans}(\{e_1 \mid \overline{s}\}]\!]^{u}_{\alpha^y_I}(\texttt{PTrans}(\texttt{groupBy}^v_{k^w}\{e^w \mid \overline{r}\})^{()}_w E)$
Deduplication plan translation rules		
U15 `dedup` in the head u is empty, w and E are non-empty		
$\texttt{PTrans}(\{\texttt{dedup}(\{e \mid \overline{r}\}) \mid p\})^{()}_w E$	$=$	$\Gamma^{\uplus/\emptyset}_e \sigma_p(\texttt{PTrans}(\{e \mid \overline{r}\})^{()}_w E)$
U16 `dedup` in the head u, w and E are non-empty		
$\texttt{PTrans}(\{\texttt{dedup}(\{e \mid \overline{r}\}) \mid p\})^{u}_w E$	$=$	$\Gamma^{\uplus/e}_u \sigma_p(\texttt{PTrans}(\texttt{dedup}(\{e^u \mid \overline{r}\}))^{()}_w E)$
U17 `dedup` in the body w and E are non-empty		
$\texttt{PTrans}(\{e_1 \mid \overline{s}, y \leftarrow \texttt{dedup}(\{e \mid \overline{r}\})\})^{u}_w E$	$=$	$\texttt{PTrans}(\{e_1 \mid \overline{s}\})^{u}_{\alpha^y_I}(\texttt{PTrans}(\texttt{dedup}\{e^w \mid \overline{r}\})^{()}_w E)$

Figure 3.4: Plan translation rules for the aggregation operations `sumBy`, `groupBy`, and `dedup`.

$\{e|\overline{r}\}$. Rules U1 - U8 in Figure 2.1 are the native bag-comprehension rules. The plan rules for key-based aggregation (`sumBy`), key-based grouping (`groupBy`), and deduplication (`dedup`) are presented in Figure 3.4. Since the key-based and deduplication operations are multiplicity-altering, the associated plan translation rules must ensure multiplicities are only adjusted within the scope of the operation without impacting that of the outer-scope. The aggregation rules in Figure 3.4 manage this by extending expressions with the attributes of the outer-scope (w or u) - denoted in the figure as superscripts on existing expressions. Each function requires a rule for handling the operation when it appears in the head or the body. The figure uses short-hand representation for head-specific rules; for example the `sumBy` in the comprehension $\{\texttt{sumBy}_k^v(e)|p\}$ would actually exist in a tuple expression: $\langle \ldots, a_i := \texttt{sumBy}_k^v, \ldots \rangle$. After translating the `sumBy` expression, the result is included within subplan E and is accessible via attribute name: $\texttt{PTrans}(\langle \ldots a_i := v \ldots \rangle)^u_{(w,v)} E$, thus handling any further translation of the tuple expression in the head. The rules for the key-based operations are now discussed.

Rules U9 - U11 handle all cases related to handling key-based sum aggregates. In general, $\texttt{sumBy}_k^v$ is translated directly to $\Gamma_k^{+/v}$; this is the translation for sum aggregates applied at the root-level when u is empty (U9). For a sum aggregate $\texttt{sumBy}_k^v(\{e|\overline{r}\})$ at a non-root level (U10), the u attributes are appended to the `sumBy` keys and to the head of the input comprehension, which is then translated following U9: $\texttt{PTrans}(\texttt{sumBy}_{k^u}^v(\{e^u|\overline{r}\}))^{()}_w$. The extension of these expressions with u ensures multiplicity information is preserved for parent-levels. The remaining comprehension is then $\texttt{PTrans}(\{(k,v)|p\})^u_w$, where (k,v) is the key-value tuple coming from `sumBy` (Section 3.1). With a non-empty u attribute, U7 concludes the subplan for this level with $\Gamma_u^{\uplus/(k,v)}$. In the case of a `sumBy` qualifier in the body of a comprehension (U11), the w attributes are appended to the `sumBy` keys and to the head of the input comprehension, which is then translated following U9: $\texttt{PTrans}(\texttt{sumBy}_{k^w}^v(\{e^w|\overline{r}\}))^{()}_w$. The extension of these expressions with w ensures no multiplicity information is lost from the current environment. This is the subplan for the upper plan translation call that now has the bound attributes from the `sumBy` qualifier in the current environment attribute set α_I^y, which includes α_I^w from the extension of the `sumBy` keys with w. The rules for key-based grouping U12 - U14 follow directly from `sumBy`, with $\Gamma^{\uplus_{gb}}$ as the operator following the plan operator semantics from Section 3.2. The key-value-grouped tuples produced by grouping are denoted $(k, _2)$ based on the default attribute name for

the newly formed group. Rules U15 - U17 handle the plan translation for the deduplication operator. The rules for $\texttt{dedup}(e)$ are derived from $\texttt{groupBy}^{\emptyset}_{\alpha^e}(e)$, applying the nest operator with an empty value parameter. The following example outlines the plan translation algorithm on the `Totals` comprehension from the introduction.

Example 1. This example walks through the comprehension to plan translation from the `Totals` example, described in NRC_{agg} syntax. The algorithm works outside-in on the comprehension applying the rules from Figure 2.1 and Figure 3.4. The corresponding plan for this example, including the output type for each operator, is shown in Figure 3.5. For readability, renaming operators are omitted from the query plan. Short-hand comprehensions $\{e|\overline{r}\}$ are presented as truncated views of lower-level comprehensions; these comprehensions and the components of the plan translation environment are tagged with level identifiers to distinguish recursive calls at various levels. At each call, a variable v is created with the type of the current attribute set, which replaces any variable reference in the current level comprehension. Bag-typed attributes later handled by unnest operators are omitted from the attribute sets.

Step 0: The plan translation starts with an empty u, w, and E.

$^0\texttt{PTrans}(\{\langle \texttt{cname} := cop.\texttt{cname}, \texttt{corders} := \{e_0|\overline{r_0}\}\rangle | cop \leftarrow \texttt{COP}\})^{()}_{()}\{()\}$

Step 1: U1 is the first rule applied, capturing the `COP` qualifier and appending a unique `copID`. The variable v_0 replaces cop in the projection on `cname`.

$^0\texttt{PTrans}(\{\langle \texttt{cname} := v_0.\texttt{cname}, \texttt{corders} := \{e_0|\overline{r_0}\}\rangle | \})^{()}_{w_0} E_0$

$w_0 = \{\texttt{copID}, \texttt{cname}\}$
$E = I(\texttt{COP})$

Step 2: With no more qualifiers in the body, U8 is triggered to handle the comprehension $\{e_0|\overline{r_0}\}$ in the head. The current environment w is passed in as the parent attributes u.

$^0\texttt{PTrans}(\{\langle \texttt{cname} := v_0.\texttt{cname}, \texttt{corders} := v_0.\texttt{corders}\rangle | \})^{()}_{w_0,\texttt{corders}}$
$^1\texttt{PTrans}(\{\langle \texttt{odate} := co.\texttt{odate}, \texttt{oparts} := \texttt{sumBy}^{\texttt{total}}_{\texttt{pname}}(\{e_1|\overline{r_1}\})\rangle |$
$\quad co \leftarrow cop.\texttt{corders}\})^{w_0}_{w_0} E_0$

$w_0 = \{\texttt{copID}, \texttt{cname}\}$
$E_0 = I(\texttt{COP})$

Step 3: U5 is triggered from a non-empty u and a path in the body, producing an outer-unnest on `corders` wrapped in an index.

$${}^{0}\texttt{PTrans}(\{\langle \texttt{cname} := v_0.\texttt{cname}, \texttt{corders} := v_0.\texttt{corders}\rangle | \})^{()}_{w_0,\texttt{corders}}$$

$${}^{1}\texttt{PTrans}(\{\langle \texttt{odate} := v_1.\texttt{odate}, \texttt{oparts} := \texttt{sumBy}^{\texttt{total}}_{\texttt{pname}}(\{e_1 | \overline{r_1}\})\rangle | \})^{u_1}_{w_1} E_1$$

$$u_1 = \{\texttt{copID}, \texttt{cname}\}$$

$$w_1 = u_1 \cup \{\texttt{coID}, \texttt{odate}\}$$

$$E_1 = I(\not{\mu}_{\texttt{corders}}\ (I(\texttt{COP})))$$

Step 4: With no more qualifiers in the body, U8 is triggered to handle the `sumBy` comprehension in the head.

$${}^{0}\texttt{PTrans}(\{\langle \texttt{cname} := v_0.\texttt{cname}, \texttt{corders} := v_0.\texttt{corders}\rangle | \})^{()}_{w_0,\texttt{corders}}$$

$${}^{1}\texttt{PTrans}(\{\langle \texttt{odate} := v_1.\texttt{odate}, \texttt{oparts} := v_1.\texttt{oparts}\}\rangle | \})^{u_1}_{w_1,\texttt{oparts}}$$

$${}^{2}\texttt{PTrans}(\texttt{odate} := v_1.\texttt{odate}, \texttt{oparts} := \texttt{sumBy}^{\texttt{total}}_{\texttt{pname}}(\{e_1 | \overline{r_1}\}) | \})^{w_1}_{w_1} E_1$$

$$w_1 = \{\texttt{copID}, \texttt{cname}, \texttt{coID}, \texttt{odate}\}$$

Step 5: U10 is triggered from the `sumBy` in the head with a non-empty u, recurring on the input comprehension and appending u to the head and key attributes. The remaining `sumBy` translation is handled in Step 9. e^s is the arithmetic expression at `total`.

$${}^{0}\texttt{PTrans}(\{\langle \texttt{cname} := v_1.\texttt{cname}, \texttt{corders} := v_0.\texttt{corders}\rangle | \})^{()}_{w_0,\texttt{corders}}$$

$${}^{1}\texttt{PTrans}(\{\langle \texttt{odate} := v_1.\texttt{odate}, \texttt{oparts} := v_1.\texttt{oparts}\}\rangle | \})^{u_1}_{w_1,\texttt{oparts}}$$

$$\Gamma^{\uplus/\langle \texttt{pname},\texttt{total}\rangle}_{u_2}(\texttt{sumBy}^{\texttt{total}}_{\{u_2,\texttt{pname}\}}({}^{2}\texttt{PTrans}(\{\langle u_2, \texttt{pname} := p.\texttt{pname}, \texttt{total} := e^s\rangle | op \leftarrow co.\texttt{oparts},$$
$$p \leftarrow \texttt{Part}, op.\texttt{pid} == p.\texttt{pid})^{()}_{w_2} E_1))$$

$$u_2 = \{\texttt{copID}, \texttt{cnamecoID}, \texttt{odate}\}$$

$$w_2 = \{\texttt{copID}, \texttt{cname}, \texttt{coID}, \texttt{odate}\}$$

Step 6: The qualifier with path $co.$`oparts` and a non-empty u triggers U5 to produce an outer-unnest wrapped in an index.

$${}^{0}\texttt{PTrans}(\{\langle \texttt{cname} := v_0.\texttt{cname}, \texttt{corders} := v_0.\texttt{corders}\rangle | \})^{()}_{w_0,\texttt{corders}}$$

$${}^{1}\texttt{PTrans}(\{\langle \texttt{odate} := v_1.\texttt{odate}, \texttt{oparts} := v_1.\texttt{oparts}\}\rangle | \})^{u_1}_{w_1,\texttt{oparts}}$$

$$\Gamma^{\uplus/\langle \texttt{pname},\texttt{total}\rangle}_{u_2}(\texttt{sumBy}^{\texttt{total}}_{\{u_2,\texttt{pname}\}}({}^{2}\texttt{PTrans}(\{\langle u_2, \texttt{pname} := v_2.\texttt{pname}, \texttt{total} := e^s\rangle |$$
$$p \leftarrow \texttt{Part}, op.\texttt{pid} == p.\texttt{pid})^{()}_{w_2} E_2)$$

$$u_2 = \{\texttt{copID}, \texttt{cname}, \texttt{coID}, \texttt{odate}\}$$

$$w_2 = u_2 \cup \{\texttt{opID}, \texttt{qty}\}$$

$$E_2 = I(\not{\mu}_{\texttt{oparts}}\ (I(\not{\mu}_{\texttt{corders}}\ (I(\texttt{COP}))))$$

Step 7: The next qualifier with a non-empty u produces an outer-join with the indexed `Part` on `pid` and a filter for the remaining conditional.

$^{0}\texttt{PTrans}(\{\langle \texttt{cname} := v_0.\texttt{cname}, \texttt{corders} := v_0.\texttt{corders}\rangle|\})^{()}_{w_0,\texttt{mutations}}$

$^{1}\texttt{PTrans}(\{\langle \texttt{odate} := v_1.\texttt{odate}, \texttt{oparts} := v_1.\texttt{oparts}\}\rangle|\})^{u_1}_{w_1,\texttt{scores}}$

$\Gamma^{\uplus/\langle \texttt{pname},\texttt{total}\rangle}_{u_2}(\texttt{sumBy}^{\texttt{total}}_{\{u_2,\texttt{pname}\}}(^{2}\texttt{PTrans}(\{\langle u_2, \texttt{pname} := v_2.\texttt{pname}, \texttt{total} := e^{s}\rangle|\})^{()}_{w_2} E_2)$

$u_2 = \{\texttt{copID}, \texttt{cname}, \texttt{coID}, \texttt{odate}\}$

$w_2 = u_2 \cup \{\texttt{opID}, \texttt{qty}, \texttt{pID}, \texttt{pname}, \texttt{price}\}$

$E_2 = \sigma_{\texttt{odate}==\text{``01:03:22''}}(E_2 ⟕_{\texttt{pid}} I(\texttt{Part}))$

Step 8: U9 handles the sum aggregate when u is non-empty and the lowest-level is finalized with U6.

$^{0}\texttt{PTrans}(\{\langle \texttt{cname} := v_0.\texttt{cname}, \texttt{corders} := v_0.\texttt{corders}\rangle|\})^{()}_{w_0,\texttt{corders}}$

$^{1}\texttt{PTrans}(\{\langle \texttt{odate} := v_1.\texttt{odate}, \texttt{oparts} := v_1.\texttt{oparts}\}\rangle|\})^{u_1}_{w_1,\texttt{oparts}}$

$\Gamma^{\uplus/\langle \texttt{pname},\texttt{total}\rangle}_{u_2}(\Gamma^{+/\texttt{total}}_{\langle u_2,\texttt{pname}\rangle}(\pi^{\langle u_2,\texttt{pname},\texttt{total}\rangle} E_2))$

$u_2 = \{\texttt{copID}, \texttt{cname}, \texttt{coID}, \texttt{odate}, \}$

$w_2 = u_2 \cup \{\texttt{opID}, \texttt{qty}, \texttt{pID}, \texttt{pname}, \texttt{price}\}$

Step 9: The first-level is finalized with U7 with u_1 non-empty.

$^{0}\texttt{PTrans}(\{\langle \texttt{cname} := v_0.\texttt{cname}, \texttt{corders} := v_0.\texttt{corders}\rangle|\})^{()}_{w_0,\texttt{corders}}$

$\Gamma^{\uplus/\langle \texttt{odate},\texttt{oparts}\rangle}_{u_1}(\Gamma^{\uplus/\langle \texttt{pname},\texttt{total}\rangle}_{u_2}(\Gamma^{+/\texttt{total}}_{\langle u_2,\texttt{pname}\rangle}(\pi^{\langle u_2,\texttt{pname},\texttt{total}\rangle} E_2)))$

Step 10: The top-level plan translation environment is finalized with U6 and the algorithm terminates returning the plan in Figure 3.5.

While the complex nature of the plan translation procedure reflects the challenges introduced by flattening, the procedure overcomes the programming mismatch by automatically generating unnest operations, null values, and unique identifiers that ensure correctness. The plans produced from the translation may be suboptimal. The algebraic form presents opportunities for further optimizations that would not be possible on the comprehension alone.

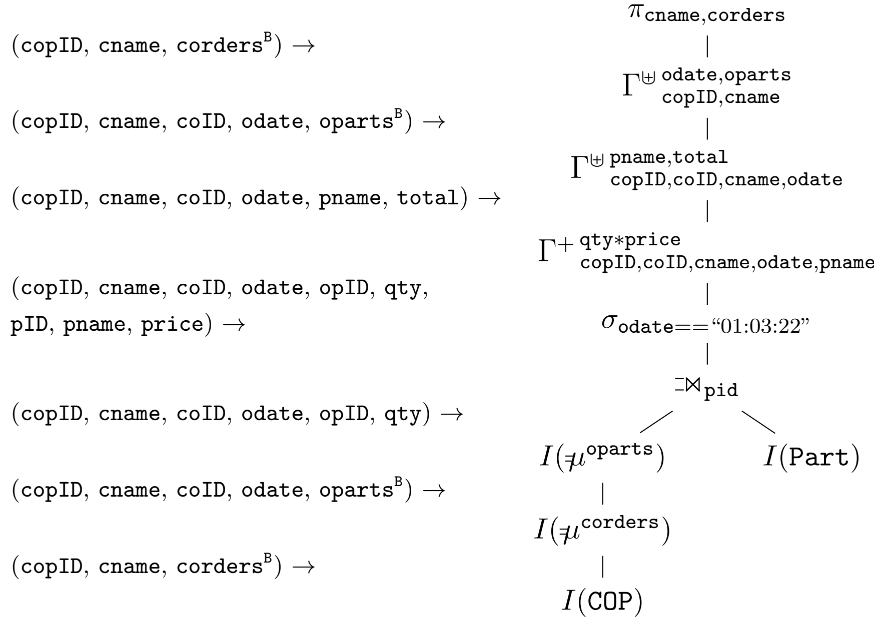


Figure 3.5: A query plan for the `Totals` example with output types. Bag types are annotated with ^{B}.

3.4 Code Generation

The query plan produced from the NRC_{agg} program is translated into a parallel data flow described in a low-level collection API via in the Code Generation Module. The code generation process is based on existing techniques specific to distributed processing platforms [60], and currently targets Apache Spark. This section discusses how code is generated from a query plan in the compilation framework, including Spark-specific design decisions.

The code generation works top-down over a query plan, translating operators of the plan language into operations over Spark Datasets. More specifically, expressions of type $Bag(R)$ are modeled as `Dataset[R]`, where `R` is a Scala case class representing a named tuple. By default, input bags modeled as Datasets are distributed at the granularity of top-level tuples. The code generation translation for all operators on Spark Datasets are presented in Figure 3.6.

Operators effect the partitioning guarantee (“partitioner” in Spark) of a Dataset. All

partitioners are key-based and ensure all values associated with the same key are assigned to the same location. Partitioning guarantees affect the amount of data movement across partitions (shuffling), which can significantly impact the execution cost. Inputs that have not been altered by an operator have no partitioning guarantee. An operator inherits the partitioner from its input and can either preserve, drop, or redefine it for the output. Join and nest operators are the only operators that directly alter partitioning since they require moving values associated with keys to the same partition. These operators control all partitioning in the standard pipeline.

Datasets verses RDDs. The generator can also use the transformations in Figure 3.7 to provide a Spark RDD implementation of the plan operators. The Dataset generator is preferred due to the benefits discussed in Section 2.6, such as the large memory foot print of using RDDs of case classes. The design choice to use Datasets was based on an initial set of experiments with the TPC-H benchmark, which revealed that an alternative encoding – using RDDs of case classes – incurs much higher memory and processing overheads. These results are discussed further in Appendix A.1.

Compiling to Notebooks. The code generator can produce both Spark applications and Apache Zeppelin notebooks. Spark applications generate a single application file that can be executed via command-line. Notebooks can be imported into the Apache Zeppelin web-interface where users are able to further interact with the outputs of the generated code. Notebook generation was designed to provide initial support for users to interface with external libraries, such as pyspark [61], scikit-learn [62], and keras [63]. The notebooks rely on Zeppelin to translate Scala Datasets into pandas dataframes for easier interaction with machine learning and other advanced statistical packages. This is merely a first step towards integrating more advanced analytics in the system.

Additional Targets. While code generation currently targets the Spark collection API, this is merely an extension of the modular framework; all components upstream of the framework are platform agnostic. A Flink code generator, for example, is currently being integrated. Other platforms such as Cascading and Apache Pig could also be supported by implementing the respective code generators.

<table>
<tr><th>Plan Operator</th><th>Definition for Dataset API</th></tr>
<tr><td>$I(X)$</td><td><pre><code>X.withColumn(index, monotonically_increasing_id())</code></pre></td></tr>
<tr><td>$\sigma_{p(x)}(X)$</td><td><pre><code>X.filter(x => p(x))</code></pre></td></tr>
<tr><td>$\pi_{a_1,\ldots,a_k}(X)$</td><td><pre><code>X.select(a1, ..., ak)</code></pre></td></tr>
<tr><td>$\mu^{a_i}(X)$</td><td><pre><code>// R does not contain x.ai
X.flatMap(x => x.ai.map(y =>
   R(x.a1, ..., x.ak, y.b1, ..., y.bj)))</code></pre></td></tr>
<tr><td>$=\!\!\!\mu^{a}(X)$</td><td><pre><code>// R does not contain x.ai
X.flatMap(x =>
  if (x.ai.isEmpty) R(x.a1, ..., x.ak, None, ..., None)
  else x.ai.map(y =>
    R(x.a1, ..., x.ak, Some(y.b1), ..., Some(y.bj))))</code></pre></td></tr>
<tr><td>$X \bowtie_{f(x)=g(y)} Y$</td><td><pre><code>X.join(Y, f === g)</code></pre></td></tr>
<tr><td>$X ⟕_{f(x)=g(y)} Y$</td><td><pre><code>X.join(Y, f === g, "left_outer")</code></pre></td></tr>
<tr><td>$\Gamma^{value(x)}_{+,\,key(x)}(X)$</td><td><pre><code>X.groupByKey(x => key(x)).agg(typed.sum(x =>
  value(x) match { case Some(v) => v; case _ => 0 }))</code></pre></td></tr>
<tr><td>$\Gamma^{value(x)}_{\uplus,\,key(x)}(X)$</td><td><pre><code>X.groupByKey(x => key(x)).mapGroups{
 case (key, values) =>
  val grp = values.flatMap{ case x =>
    value(x) match {
     case Some(t) => Seq(t)
     case _ => Seq() }
   }.toSeq
  (key, grp)
}</code></pre></td></tr>
</table>

Figure 3.6: Plan language operators and their semantics for Spark Datasets.

Plan Operator	Definition for RDD API
$I(X)$	`X.map(x => addIndex(x)) // addIndex adds an index to tuple x`
$\sigma_{p(x)}(X)$	`X.filter(x => p(x))`
$\pi_{a_1,\ldots,a_k}(X)$	`X.map(x => R(x.a1, ..., x.ak))`
$\mu^{a_i}(X)$	`// x.drop(a) excludes attribute a from tuple x` `X.flatMap(x => x.a.map(y => (x.drop(a), y)))`

$\not\mu^{a}(X)$	`// x.drop(a) excludes attribute a from x`
	`X.flatMap(x => if (x.a.isEmpty) Vector((x, null))`
	`  else x.a.map(y => (x.drop(a), y)))`

$X \bowtie_{f(x)=g(y)} Y$	`val keyX = X.map(x => (f(x), x))`
	`val keyY = Y.map(y => (f(y), y))`
	`keyX.join(keyY).values`

$X ⟕_{f(x)=g(y)} Y$	`val keyX = X.map(x => (f(x), x))`
	`val keyY = Y.map(y => (f(y), y))`
	`keyX.leftOuterJoin(keyY).values`

$\Gamma^{value(x)}_{+,\, key(x)}(X)$	`X.map(x => if (value(x) != null) (key(x), value(x))` `  else (key(x), 0)).reduceByKey(_+_)`

$\Gamma^{value(x)}_{\uplus,\, key(x)}(X)$	`X.map(x => if (value(x) != null){`
	`    (key(x), Vector(value(x)))`
	`  } else (key(x), Vector())).reduceByKey(_++_)`

Figure 3.7: Plan language operators and their semantics for Spark RDDs.

Chapter 4

Query and Data Shredding

This chapter presents the shredding techniques of TraNCE that overcome the scalability issues of nested representations on distributed processing frameworks. The extensions of NRC_{agg} to support shredding are first introduced in Section 4.1. The query shredding algorithm described in Section 4.2 takes a two-step approach, first adapting the symbolic shredding discussed in Section 2.5.3 to NRC_{agg}, and then applying a novel materialization algorithm to return explicit, materialized dictionaries designed for sequential execution. The materialized program is then optimized with the shredding-specific optimizations described in Section 4.3. These sections will use the `Totals` example throughout following the recursive view; Section 4.4 presents the resulting translation in relational view.

Shredding is an optimization within the compilation framework that the user never interacts with directly. A user submits an NRC_{agg} query to the framework and the compiler generates the code for executing the shredded query as a distributed application. Thus, the *shredded compilation route* extends the standard compilation route with the shredding techniques discussed in this chapter. The extensions to support shredding in the compilation framework are presented in the remaining sections, including the shredded `Totals` query through plan translation and code generation.

Recall that the `Totals` example operates on the `COP` dataset that is a collection of

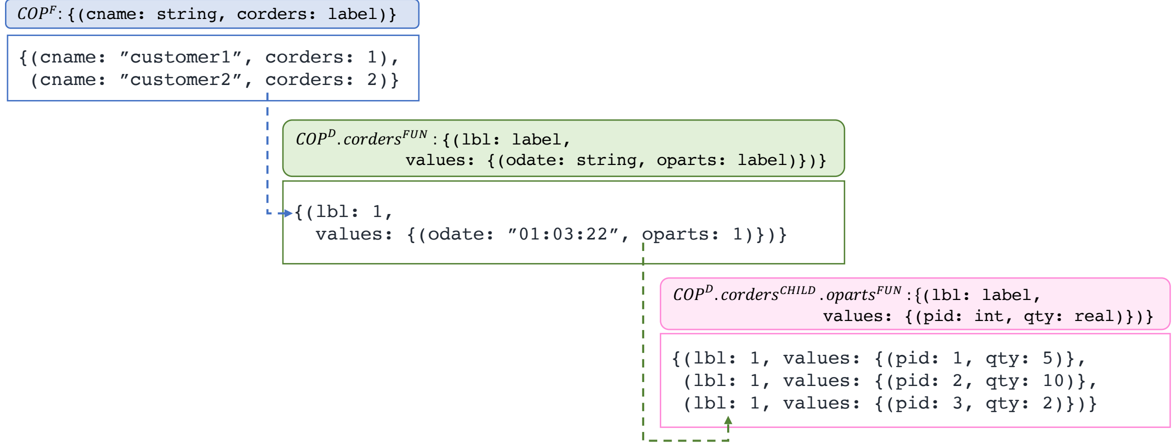


Figure 4.1: The shredded data of COP encoded as label/bag pairs (recursive view) with one top-level $\texttt{COP}^{\mathrm{F}}$ and two value dictionaries $\texttt{COP}^{\mathrm{F}}_{\texttt{corders}}$ and $\texttt{COP}^{\mathrm{F}}_{\texttt{corders_oparts}}$

.

nested objects. The type of COP is:

$$\texttt{COP} : \mathit{Bag}\,(\langle\,\texttt{cname} : \mathit{string}, \texttt{corders} : \mathit{Label}\rangle)\; \mathit{Bag}\,(\langle\,\texttt{odate} : \mathit{date}, \texttt{oparts} : \mathit{Bag}\,(\langle\,\texttt{pid} : \mathit{int}, \texttt{qty} : \mathit{real}\rangle)\,\rangle)\,\rangle).$$

The shredded representation of COP has a top-level flat bag $\texttt{COP}^{\mathrm{F}}$ of type:

$$\texttt{COP}^{\mathrm{F}} : \mathit{Bag}(\langle\,\texttt{cname} : \mathit{string}, \texttt{corders} : \mathit{Label}\rangle)$$

and a dictionary tree of $\texttt{COP}^{\mathrm{D}}$ of type:

$$\begin{aligned}\texttt{COP}^{\mathrm{D}} : \langle\ & \texttt{corders}^{\texttt{fun}} : \mathit{Label} \to \mathit{Bag}(\langle\,\texttt{odate} : \mathit{string}, \texttt{oparts} : \mathit{Label}\rangle),\\ & \texttt{corders}^{\texttt{child}} : \mathit{Bag}(\langle\\ & \quad \texttt{oparts}^{\texttt{fun}} : \mathit{Label} \to \mathit{Bag}(\langle \texttt{pid} : \mathit{int}, \texttt{qty} : \mathit{real}\rangle),\\ & \quad \texttt{oparts}^{\texttt{child}} : \mathit{Bag}(\langle\rangle)\rangle)\rangle)\end{aligned}$$

The $\texttt{COP}^{\mathrm{D}}$ dictionary tree encodes a $\texttt{corders}^{\texttt{fun}}$ dictionary associated to corders labels, a $\texttt{oparts}^{\texttt{fun}}$ dictionary associated to oparts labels, and the nesting structure via the child attributes.

The shredded COP data is provided in Figure 4.1 using the recursive view. Here, there is one top-level $\texttt{COP}^{\mathrm{F}}$ and two value dictionaries $\texttt{COP}^{\mathrm{F}}.\texttt{corders}^{\texttt{fun}}$ and

$$
\begin{aligned}
e ::= & \textit{[Following Figure 3.1]} \\
& \mid \texttt{Label}(var, \ldots) \mid \texttt{match}\ e = \texttt{Label}(var, \ldots)\ \texttt{then}\ e \\
& \mid \texttt{Lookup}(e, e) \mid \lambda var.e \\
& \mid e\ \texttt{DictTreeUnion}\ e \\
& \mid \langle \ldots, a_i : \langle a_i^{\texttt{fun}} : e, a_i^{\texttt{child}} : e \rangle, \ldots \rangle \\
& \boxed{\begin{array}{l} \mid \texttt{MatLookup}(e, e) \\ \mid \texttt{BagToDict}(e) \end{array}}
\end{aligned}
$$

Figure 4.2: The syntax of $\text{NRC}_{agg}^{Lbl+\lambda}$.

$$
\begin{aligned}
[\![\texttt{MatLookup}(e_1, e_2)]\!]_p &= [\![e_1]\!]_p([\![e_2]\!]_p) \\
[\![\texttt{BagToDict}(\{\langle \texttt{lbl} := l, \texttt{value} := \{\langle \overline{a} \rangle | e\} \rangle | l \leftarrow L\})]\!]_p &= \biguplus_{l \in [\![L]\!]_p} \biguplus_{[\![e]\!]_p} \{\ [\![\langle \texttt{lbl} := l, \overline{a} \rangle]\!]_p \} \\
& \textit{where}\ \overline{a} = a_1 : e_1, \ldots, a_k : e_k
\end{aligned}
$$

Figure 4.3: Semantics of $\text{NRC}_{agg}^{Lbl+\lambda}$, following Figure 2.3.

$\texttt{COP}^{\text{F}}.\texttt{corders}^{\texttt{child}}.\texttt{oparts}^{\texttt{fun}}$. The flat bag `Part` is also used by the `Totals` program. The shredded representation of `Part` contains only a top-level $\texttt{Part}^{\text{F}}$ that is equivalent to `Part` and an empty dictionary, ie. $\texttt{Part}^{\text{D}} = \langle \rangle$. Section 4.4 will discuss how the alternative relational view is used within the framework.

4.1 Shredded NRC

The intermediate language that extends NRC_{agg} to support the shredding transformation is denoted $\text{NRC}_{agg}^{Lbl+\lambda}$. The syntax of $\text{NRC}_{agg}^{Lbl+\lambda}$ is provided in Figure 4.2, which follows $\text{NRC}^{Lbl+\lambda}$ (Section 2.5.3) with a few additional primitives outlined. The semantics for the additional primitives are provided in Figure 4.3.

The `MatLookup` operation logically separates lookup operations on symbolic dictionaries from materialized dictionaries to support the TraNCE materialization strategy.

Whereas a `Lookup` is generated by a symbolic shredding procedure to operate on a symbolic dictionary, a `MatLookup` is generated by the materialization procedure to operate on explicit, materialized dictionary expressions.

The `BagToDict` operation casts a dictionary of label/bag pairs in recursive view (Section 2.5.2) to relational view, ie. to a collection of flat tuples with `lbl` attributes. This allows for flexible representation of a materialized dictionary; since the recursive representation is not truly flat, the relational view can be more optimal for later stages of compilation. The input of `BagToDict` is restricted to an expression of type $Bag\,(\langle\, \texttt{lbl}\colon Label,\ \texttt{value}\colon\ Bag\,\rangle)$.

4.2 Query Shredding Algorithm

The shredding transformation takes an NRC_{agg} query and outputs a shredded $\mathrm{NRC}_{agg}^{Lbl+\lambda}$ query. Following from Section 2.5.3, the query shredding algorithm takes as input an expression e of type T and produces an expression e^{F} of type T^{F} and a dictionary tree e^{D} of tuple type T^{D}. The algorithm transforms the evaluation problem for e to evaluation problems for e^{F} and e^{D}: when the output of e on an input i is o, then the outputs of e^{F} and e^{D} on the shredded representation of i will be o^{F} and o^{D}, respectively.

The shredding transformation consists of two phases, described in detail in the next two sections. The first is the *symbolic query shredding* phase, which produces succinct, symbolic expressions manipulating intermediate and output symbolic dictionaries defined via λ expressions. This phase adapts the work from [20] to the NRC_{agg} source language. The key to the succinctness of the representation is that the dictionaries are not presented explicitly as a collection of label/bag pairs, but rather are *over-approximated* by λ-terms that provide general recipes for computing a bag from an arbitrary label. In the second phase, the *materialization transformation* takes the symbolic dictionary expressions and produces a sequence of materialized dictionary expressions that are free of λ abstractions and are explicitly represented as a collection of label/bag pairs.

	Pattern of e		$\mathcal{F}(e)$	$\mathcal{D}(e)$
	...Following Figure 2.4			
5	$\langle \dots a_i := e_i, \dots$	e_i of bag type	$\langle \dots a_i := \texttt{Label}((x^{\text{F}})_{x \in \text{FreeVar}(e_i)}), \dots$	$\langle \dots a_i^{\texttt{fun}} := \lambda l.\ \texttt{match}\ l = \texttt{Label}(\dots)\ \texttt{then}\ \mathcal{F}(e_i),$ $a_i^{\texttt{child}} := \{\mathcal{D}(e_i)\}, \dots$
	...			
14	$\texttt{get}(e)$		$\texttt{get}(\mathcal{F}(e))$	$\mathcal{D}(e)$
15	$\texttt{udf}(e)$		$\mathcal{F}(\texttt{udf})(\mathcal{F}(e))$	$\mathcal{D}(\texttt{udf})(\mathcal{D}(e))$
16	$\texttt{dedup}(e)$		$\texttt{dedup}(\mathcal{F}(e))$	$\mathcal{D}(e)$
17	$\texttt{sumBy}_{key}^{value}(e)$		$\texttt{sumBy}_{key}^{value}(\mathcal{F}(e))$	$\mathcal{D}(e)$
18	$\texttt{groupBy}_{key}^{value}(e)$		$\texttt{dedup}(\mathcal{F}(e'))$	$\mathcal{D}(e')$
			e' = $\{\langle \texttt{key} := k.key, \texttt{value} := \{\langle \texttt{value} := v.value\rangle \mid v \leftarrow e, v.key == k.key\}\rangle \mid k \leftarrow e\}$	

Figure 4.4: Query shredding algorithm following from Figure 2.4 with aggregation extensions.

4.2.1 Symbolic query shredding

The symbolic query shredding algorithm from Section 2.5.3 is extended to support the additional operators of $\text{NRC}_{agg}^{Lbl+\lambda}$. Figure 4.4 shows the extensions to the shredding transformation in terms of recursive functions $\mathcal{F}$ and $\mathcal{D}$, following from Figure 2.4. Given a source expression e, the invocation $\mathcal{F}(e)$ returns the expression e^{F} computing the flat version of the output, while $\mathcal{D}(e)$ returns the dictionary tree e^{D} corresponding to e^{F}.

The shredding algorithm is refined to retain only the relevant attributes of free variables in a `Label` (line 5), thus contributing to a succinct representation; For the remaining constructs, the functions $\mathcal{F}$ and $\mathcal{D}$ proceed recursively, maintaining the invariant that for any source expression e, the free variables of e^{F} or e^{D} in the shredded representation correspond to the free variables of e. The interesting case is for the expression case $\texttt{E} = \texttt{groupBy}_{key}^{value}(e)$. The `groupBy` operator is a primitive that takes a flat bag e; simplifying to *key* and *value* of only a single attributes `key` and `value`, this can be alternatively expressed in NRC_{agg} as:

$\texttt{Keys} \Leftarrow \texttt{dedup}(\texttt{for}\ k\ \texttt{in}\ e\ \texttt{union}\{\langle \texttt{key} := k.\texttt{key}\rangle\})$

```
E ⇐
for k in Keys union
  {⟨key := k.key, value :=
    for v in e union if v.key == k.key then
        {⟨value := v.value⟩}
```

The symbolic shredding of the above produces:

```
Keys^F ⇐ dedup(for k^F in e^F union{⟨key := k.key⟩})

E^F ⇐
 for k^F in Keys^F union
  {⟨key := k^F.key, value := Label_value(k^F.key)⟩}

E^D ⇐
 ⟨value^fun : λl.match l = Label_value(k^F.key) in
   for v in e union if v.key == k.key then
     {⟨value := v.value⟩}
  value^child : {⟨⟩}⟩
```

The transformation can be achieved in fewer steps by delaying the dedup until the shredding transformation:

```
E^F ⇐
 dedup(for k^F in Keys^F union
  {⟨key := k^F.key, value := Label_value(k^F.key)⟩})

E^D ⇐
 ⟨value^fun : λl.match l = Label_value(k^F.key) in
   for v in e union if v.key == k.key then
     {⟨value := v.value⟩}
  value^child : {⟨⟩}⟩
```

The algorithm achieves this by rewriting the groupBy denoted by the NRC_{agg} expression e' in Figure 4.4, and wrapping the flat component in a dedup operator. The symbolic shredding algorithm is now presented on the Totals program.

Example 2. Here the shredding algorithm is exhibited on the Totals example. The algorithm produces a query $\texttt{Totals}^{\mathrm{F}}$ computing the top-level flat bag and a query $\texttt{Totals}^{\texttt{Dict}}$ computing the dictionary tree for $\texttt{Totals}^{\mathrm{F}}$.

Totals is matched to the for construct in F and the process recurs to derive

$\texttt{Totals}^{\mathrm{F}}$. The shredded query returning the top-level bag, $\texttt{Totals}^{\mathrm{F}}$, is:

$$\texttt{for}\ cop^{\mathrm{F}}\ \texttt{in}\ \texttt{COP}^{\mathrm{F}}\ \texttt{union}$$
$$\{\,\langle\,\texttt{cname} := cop^{\mathrm{F}}.\texttt{cname},\ \texttt{corders} := \texttt{Label}_{\texttt{corders}}(cop^{\mathrm{F}}.\texttt{corders})\,\rangle\,\}$$

Recall that $\texttt{COP}^{\mathrm{F}}$ is a flat bag, so $\texttt{Totals}^{\mathrm{F}}$ indeed computes a bag of flat tuples. The labels $\texttt{Label}_{\texttt{corders}}$ encapsulate only the required information for downstream; ie. $cop^{\mathrm{F}}.\texttt{corders}$ is the only expression needed in the subsequent dictionary expression. Unused `let` bindings to cop^{D} are dropped. $\texttt{Totals}^{\mathrm{D}}$ is derived after matching `corders` in the top-level tuple constructor of `Totals` (line 5):

$$\begin{array}{l}
\texttt{let}\ cop^{\mathrm{D}} := \texttt{COP}^{\mathrm{D}}\ \texttt{in} \\
\langle \texttt{corders}^{\texttt{fun}} := \lambda l.\ \texttt{match}\ l = \texttt{Label}_{\texttt{corders}}(cop^{\mathrm{F}}.\texttt{corders})\ \texttt{then} \\
\quad \texttt{for}\ co^{\mathrm{F}}\ \texttt{in}\ \texttt{Lookup}(cop^{\mathrm{D}}.\texttt{corders}^{\texttt{fun}}, cop^{\mathrm{F}}.\texttt{corders})\ \texttt{union} \\
\quad\quad \{\,\langle\,\texttt{odate} := co^{\mathrm{F}}.\texttt{odate},\ \texttt{oparts} := \texttt{Label}_{\texttt{oparts}}(co^{\mathrm{F}}.\texttt{oparts}, co^{\mathrm{F}}.\texttt{odate})\,\rangle\,\}, \\
\texttt{corders}^{\texttt{child}} := \{\texttt{Totals}^{\mathrm{D}}_{\texttt{corders}}\}\,\rangle
\end{array}$$

The unused `let` binding to co^{D} in the expression computing the dictionary $\texttt{corders}^{\texttt{fun}}$ is omitted. The query producing the lowest-level dictionary tree $\texttt{Totals}^{\mathrm{D}}_{\texttt{corders}}$ is:

$$\begin{array}{l}
\texttt{let}\ co^{\mathrm{D}} := \texttt{get}(cop^{\mathrm{D}}.\texttt{corders}^{\texttt{child}})\ \texttt{in} \\
\langle\,\texttt{oparts}^{\texttt{fun}} := \lambda l.\ \texttt{match}\ l = \texttt{Label}_{\texttt{oparts}}(co^{\mathrm{F}}.\texttt{oparts}, co^{\mathrm{F}}.\texttt{odate})\ \texttt{then} \\
\quad \texttt{sumBy}^{\texttt{total}}_{\texttt{pname}}(\\
\quad\ \ \texttt{for}\ op^{\mathrm{F}}\ \texttt{in}\ \texttt{Lookup}(co^{\mathrm{D}}.\texttt{oparts}^{\texttt{fun}}, co^{\mathrm{F}}.\texttt{oparts})\ \texttt{union} \\
\quad\quad \texttt{for}\ p^{\mathrm{F}}\ \texttt{in}\ \texttt{Part}^{\mathrm{F}}\ \texttt{union} \\
\quad\quad\ \ \texttt{if}\ (p^{\mathrm{F}}.\texttt{pid} == op^{\mathrm{F}}.\texttt{pid}\ \ \&\&\ \ co^{\mathrm{F}}.\texttt{odate} == \text{“01 : 03 : 22”})\ \texttt{then} \\
\quad\quad\quad \{\,\langle\,\texttt{pname} := p^{\mathrm{F}}.\texttt{pname},\ \texttt{total} := op^{\mathrm{F}}.\texttt{qty} * p^{\mathrm{F}}.\texttt{price}\,\rangle\,\}), \\
\texttt{oparts}^{\texttt{child}} := \{\langle\rangle\,\}\,\rangle
\end{array}$$

4.2.2 Materialization

The symbolic dictionaries produced by the first phase of the shredding algorithm are useful in keeping the inductively-formed expressions succinct, but do not provide an explicit dictionary encoding. The second phase of the shredding procedure uses materialization to translate the symbolic dictionaries into materialized dictionaries that eliminate λ-terms and produce an explicit representation of the shredded output. In the case of a query with flat output, materialization will be a variant of well-known normalization and conservativity algorithms [10, 64]. In the case of nested output, materialization will be the key distinguishing feature from monolithic shredding (Section 2.5.3). The *sequential materialization* approach produces a sequence of queries, one

$\textsc{Materialize}$(expression e^{F}, expression e^{D}, var $\texttt{Top}$, funct *resolver*)

1. $e_1^{\mathrm{F}} = \textsc{ReplaceSymbolicDicts}(e^{\mathrm{F}}, \mathit{resolver})$ //by materialized dicts
2. $\textsc{Emit}(\texttt{Top} \Leftarrow e_1^{\mathrm{F}})$
3. $e_1^{\mathrm{D}} = \textsc{Normalize}(e^{\mathrm{D}})$ // recursively inline `let` bindings
4. $\textsc{MaterializeDict}(e_1^{\mathrm{D}}, \texttt{Top}, \mathit{resolver})$

$\textsc{MaterializeDict}$(expression e^{D}, var $\texttt{Parent}_{index}$, funct *resolver*)

```
1  switch e^D:
2    case ⟨a_1 := e_1, ..., a_n := e_n⟩ ⇒
3      foreach a^fun ∈ {a_1, ..., a_n}
4        Emit(LabDomain_{index_a} ⇐
5            dedup(for x in Parent_{index} union {⟨lbl := x.a⟩}))
6        fun = ReplaceSymbolicDicts(e^D.a^fun, resolver)
7        Emit(MatDict_{index_a} ⇐
8            for l in LabDomain_{index_a} union
9              {⟨lbl := l.lbl, value := fun(l.lbl)⟩})
10       resolver = AddMapping(resolver, e^D.a^fun, MatDict_{index_a})
11       MaterializeDict(get(e^D.a^child), MatDict_{index_a}, resolver)
12   case DictTreeUnion(e_1^D, e_2^D) ⇒
13     MaterializeDict(e_1^D, Parent_{index}, resolver)
14     MaterializeDict(e_2^D, Parent_{index}, resolver)
```

Figure 4.5: Materialization algorithms.

for each level of the output, with the query for a given dictionary D depending on the output of its parent. The advantage is that the labels of D will already be materialized in the parent object, which avoids the duplication of computation and repeated iteration artifacts of monolithic shredding.

The materialization phase produces a sequence of assignments, given a shredded expression and its dictionary tree. For each symbolic dictionary Q^{D}_{index} in the dictionary tree, the transformation creates one assignment $\texttt{MatDict}_{index} \Leftarrow e$, where the expression e computes a bag of tuples representing the materialized form of Q^{D}_{index}. Each assignment can depend on the output of the prior ones. The strategy works downward on the dictionary tree, keeping track of the assigned variable for each symbolic dictionary. Prior to producing each assignment, the transformation rewrites the expression e to replace any symbolic dictionary by its assigned variable and any `Lookup` on a symbolic dictionary by a `MatLookup` on its materialized variant.

Materialization needs to resolve the domain of a symbolic dictionary. In our baseline materialization, the expression e computing $\texttt{MatDict}_{index}$ takes as input a *label domain*, the set of labels produced in the parent assignment. The expression e then simply iterates over the label domain and evaluates the symbolic dictionary Q^{D}_{index} for each label.

The materialization algorithm from Figure 4.5 produces a sequence of assignments given a shredded expression, its dictionary tree, and a variable to represent the top-level expression. The Materialize procedure first replaces the (input) symbolic dictionaries in e^{F} by their materialized counterparts (line 1) and then assigns this rewritten expression to the provided variable (line 2). Prior to traversing the dictionary tree, its dictionaries are simplified by inlining each `let` binding produced by the shredding algorithm (line 3).

The MaterializeDict procedure performs a depth-first traversal of the dictionary tree. For each label-valued attribute a, the assignment computing the set of labels produced in the parent assignment is first emitted (lines 3-4). The dictionary is then rewritten to $a^{\texttt{fun}}$ to replace all references to symbolic dictionaries, each of them guaranteed to have a matching assignment since the traversal is top-down. A final assignment is then produced that computes a bag of label-value tuples representing the materialized form of $a^{\texttt{fun}}$ (lines 6-8), producing the domain of labels before recurring to

the child dictionary tree (line 9).

The ReplaceSymbolicDicts function first recursively inlines all let bindings in the given expression and then performs the following actions: 1) replaces any `Lookup` over an input symbolic dictionary with a `MatLookup` over the corresponding materialized dictionary obtained using the *resolver* function; 2) β-reduces any `Lookup` over an intermediate symbolic dictionary (lambda), returning its body expression with the given label inlined; and 3) replaces any `Lookup` over a `DictTreeUnion` with a union of two recursively resolved lookups.

The materialization algorithm is now presented on the `Totals` example.

Example 3. This example exhibits the Materialize procedure on the shredded queries of the `Totals` program $\texttt{Totals}^{\mathrm{F}}$ and $\texttt{Totals}^{\mathrm{D}}$ from Example 2. As with symbolic shredding, this example contains simple post-processing, including trivial inline and dropping unused `let` bindings. The materialization sequence first computes the flat bag, which is equivalent to $\texttt{Totals}^{\mathrm{F}}$ and stores the labels occurring in it in a domain.

Let Mat denote the materialization of the top-level input bags $\texttt{COP}^{\mathrm{F}}$ and $\texttt{Part}^{\mathrm{F}}$, and the two symbolic dictionaries from $\texttt{COP}^{\mathrm{D}}$. The procedure first replaces $\texttt{COP}^{\mathrm{F}}$ by MatCOP in $\texttt{Totals}^{\mathrm{F}}$, followed by emitting the assignment to the provided variable TopBag:

$$\begin{aligned}&\text{TopBag} \Leftarrow \texttt{for } cop^{\mathrm{F}} \texttt{ in } \text{MatCOP} \texttt{ union}\\&\qquad\{\langle \texttt{cname} := cop^{\mathrm{F}}.\texttt{cname}, \texttt{corders} := \texttt{Label}_{\texttt{corders}}(cop^{\mathrm{F}}.\texttt{corders})\rangle\}\end{aligned}$$

MaterializeDict produces an assignment computing the set of `corders` labels from the parent TopBag (lines 3-4):

$$\begin{aligned}&\text{LabDomain}_{\texttt{corders}} \Leftarrow\\&\quad \texttt{dedup}(\texttt{for } x \texttt{ in } \text{TopBag} \texttt{ union } \{\langle \texttt{lbl} := x.\texttt{corders}\rangle\})\end{aligned}$$

The function uses the labels from $\text{LabDomain}_{\texttt{corders}}$ to materialize the $\texttt{corders}^{\texttt{fun}}$ dictionary:

$$\begin{aligned}&\text{MatDict}_{\texttt{corders}} \Leftarrow\\&\quad \texttt{for } l \texttt{ in } \text{LabDomain}_{\texttt{corders}} \texttt{ union}\\&\quad \{\langle \texttt{lbl} := \texttt{l.lbl},\\&\qquad \texttt{value} := \texttt{match } l.\texttt{lbl} = \texttt{Label}_{\texttt{corders}}(cop^{\mathrm{F}}.\texttt{corders}) \texttt{ then}\\&\qquad\quad \texttt{for } co^{\mathrm{F}} \texttt{ in } \texttt{MatLookup}(\text{MatCOP}_{\texttt{corders}}, cop^{\mathrm{F}}.\texttt{corders}) \texttt{ union}\\&\qquad\qquad \{\langle \texttt{odate} := co^{\mathrm{F}}.\texttt{odate}, \texttt{oparts} := \texttt{Label}_{\texttt{oparts}}(co^{\mathrm{F}}.\texttt{oparts}, co^{\mathrm{F}}.\texttt{odate})\rangle\}\end{aligned}$$

The query computing `value` corresponds to the body of the $\texttt{corders}^{\texttt{fun}}$ dictionary, with the `Lookup` and its symbolic dictionary replaced by their materialized counterparts (line 5). The function finally recurs on the dictionary tree for `oparts`, deriving the label domain for $\texttt{oparts}^{\texttt{fun}}$ from $\textsc{MatDict}_{\texttt{corders}}$.

$$
\begin{array}{l}
\textsc{LabDomain}_{\texttt{corders_oparts}} \Leftarrow \\
\quad \texttt{dedup}\big(\texttt{for}\ x\ \texttt{in}\ \textsc{MatDict}_{\texttt{corders}}\ \texttt{union}\ \{\langle\, \texttt{lbl} := \texttt{x.oparts}\,\rangle\}\big)
\end{array}
$$

$$
\begin{array}{l}
\textsc{MatDict}_{\texttt{corders_oparts}} \Leftarrow \\
\quad \texttt{for}\ l\ \texttt{in}\ \textsc{LabDomain}_{\texttt{corders_oparts}}\ \texttt{union} \\
\quad\ \{\,\langle\, \texttt{lbl} := \texttt{l.lbl}, \\
\qquad \texttt{value} := \texttt{match}\ l.\texttt{lbl} = \texttt{Label}_{\texttt{oparts}}(co^{\text{F}}.\texttt{oparts}, co^{\text{F}}.\texttt{odate})\ \texttt{then} \\
\qquad\ \texttt{sumBy}^{\texttt{total}}_{\texttt{pname}}(\\
\qquad\quad \texttt{for}\ op^{\text{F}}\ \texttt{in}\ \texttt{MatLookup}\big(\textsc{MatCOP}_{\texttt{corders_oparts}}, co^{\text{F}}.\texttt{oparts}\big)\ \texttt{union} \\
\qquad\qquad \texttt{for}\ p^{\text{F}}\ \texttt{in}\ \texttt{Part}^{\text{F}}\ \texttt{union} \\
\qquad\qquad\quad \texttt{if}\ \ (p^{\text{F}}.\texttt{pid} == op^{\text{F}}.\texttt{pid}\ \ \&\&\ \ co^{\text{F}}.\texttt{odate} == \text{“}01:03:22\text{”})\ \texttt{then} \\
\qquad\qquad\qquad \{\,\langle\, \texttt{pname} := p^{\text{F}}.\texttt{pname}, \texttt{total} := op^{\text{F}}.\texttt{qty} * p^{\text{F}}.\texttt{price}\,\rangle\,\}\,)\,\rangle\,\}
\end{array}
$$

In the lowest-level dictionary expression $\textsc{MatDict}_{\texttt{corders_oparts}}$, the `sumBy` can operate directly over the $\texttt{COP}^{\text{D}}_{\texttt{corders_oparts}}$ value dictionary. This produces a plan that is a subset of the plan in Figure 3.5, isolating the join with `Part` and the aggregate operation to a single value dictionary. This is a feature of the shredded representation referred to as a *localized* operation, where nested operations can be applied directly to value dictionaries relevant to that level.

4.2.3 Discussion of Sequential Shredding

While monolithic shredding constructs stand-alone queries that can be executed independently on-top of a relational engine, the sequential approach follows the execution strategy distributed processing systems. The sequential approach also maximizes the sharing opportunities in the output, since a unique label identifying a nested object may be referenced many times in the parent. Sequential shredding, however, introduces deduplication and lookup operations that can lead to data shuffle.

In the case of sequential evaluation, the decision to inline `let` bindings can have performance impacts that are currently only controlled by user-based decisions. Consider a single query that builds up intermediate nested objects with `let` bindings. These bindings will be duplicated across each of the materialized dictionary expressions and

then inlined, which removes any possibility that these can be shared. If instead these intermediate nested objects are built up using assignment, then the resulting dictionary expressions are marked for materialization and can thus benefit from sharing. The choice to automatically inline lets is inherited from standard normalization and is done to guarantee that no intermediate materializations exist in the resulting dictionary expressions. On the other hand, the decision to mark an expression for materialization will restrict the optimizations that can happen downstream in the query planning module. As noted in Chapter 1, we focus on analysis pipelines where the intermediate nested outputs are important due to reuse. Materialization is thus done as a heuristic to benefit these shared-transformation pipelines; however, the choice to inline verse materialize is a complicated, performance impacting decision that should be controlled by internal optimizations and not at the users delegation. The importance of this design decision is noted and is currently a primary focus in ongoing work.

4.3 Domain Elimination

In many cases the materialization algorithm produces label domains that are redundant. This represents one major shortcoming of sequential shredding: in some cases making use of labels materialized in the parent is unnecessary. In order to detect such situations, the materialization procedure is optimized using *domain elimination* rules. The first rule recognizes that a child symbolic dictionary is an expression of the form:

$$\lambda l.\, \texttt{match}\ l = \texttt{Label}(x)\ \texttt{then} \\ \quad \texttt{for}\ y\ \texttt{in}\ \texttt{Lookup}(D, x.a)\ \texttt{union}\ e$$

where the only used attribute of x is a of label type. Computing the domain of labels for this dictionary can be skipped and instead the label-value pairs can be computed directly from the materialized dictionary $\texttt{MatD}$ corresponding to D:

$$\texttt{for}\ z\ \texttt{in}\ \texttt{MatD}\ \texttt{union} \\ \{\langle \texttt{lbl} := z.\texttt{lbl}, \\ \texttt{value} := \texttt{let}\ x := \langle a := z.\texttt{lbl} \rangle\ \texttt{in} \\ \texttt{for}\ y\ \texttt{in}\ z.\texttt{value}\ \texttt{union}\ e \rangle\}$$

This rule is beneficial when the size of the label domain from the parent assignment is comparable to the size of $\texttt{MatD}$. This rule can also be extended to match a `sumBy`

construct around the `for` construct.

The second rule for domain elimination recognizes when a label l encodes a non-label attribute b filtering a bag:

$$\lambda l.\, \texttt{match}\ l = \texttt{Label}(x)\ \texttt{then}$$
$$\quad \texttt{for}\ y\ \texttt{in}\ Y\ \texttt{union}\ \texttt{if}\ (y.a == x.b)\ \texttt{then}\ e$$

If no other attribute of x appears in e, the label-value pairs can be produced from Y using the value of $y.a$:

$$\texttt{groupBy}_{\texttt{lbl}}\big(\texttt{for}\ y\ \texttt{in}\ Y\ \texttt{union}$$
$$\qquad \texttt{let}\ x := \langle b := y.a\rangle\ \texttt{in}$$
$$\qquad \{\,\langle\,\texttt{lbl} := \texttt{Label}(x), \texttt{value} := e\,\rangle\,\}\big)$$

This rule transforms the variable x from free to bound, allowing the materialized dictionary to be computed from Y only.

Example 4. Returning to Example 3, the application of the second rule avoids producing $\text{LabDomain}_{\texttt{corders}}$ as an intermediate result. The materialization of $\texttt{corders}^{\texttt{fun}}$ now corresponds to:

$$\text{MatDict}_{\texttt{corders}} \Leftarrow$$
$$\quad \texttt{for}\ z\ \texttt{in}\ \text{MatCOP}_{\texttt{corders}}\ \texttt{union}$$
$$\quad \{\,\langle\,\texttt{lbl} := z.\texttt{lbl},$$
$$\qquad \texttt{value} := \texttt{for}\ co^{\text{F}}\ \texttt{in}\ z.\texttt{value}\ \texttt{union}$$
$$\qquad\qquad \{\langle\,\texttt{odate} := co^{\text{F}}.\texttt{odate},\ \texttt{oparts} := \texttt{Label}(co^{\text{F}}.\texttt{oparts}, co^{\text{F}}.\texttt{odate})\,\rangle\}$$

Since $\text{MatDict}_{\texttt{corders_oparts}}$ depends on $co^{\text{F}}.\texttt{odate}$, the domain of labels is required and no further domain optimizations can be applied.

4.4 Relational View

Dictionaries produced from the shredding transformation are encoded using the recursive view. This follows the succinctness of the symbolic transformation, where dictionary types are $Label \to Bag$ as discussed in Section 2.5.2. This gives value dictionaries and materialized dictionary expressions of type $Bag(\langle \texttt{lbl} : Label, \texttt{value} : Bag(T^{\text{F}})\rangle)$; however, this encoding is not truly flat and may not lead to optimal distribution. To

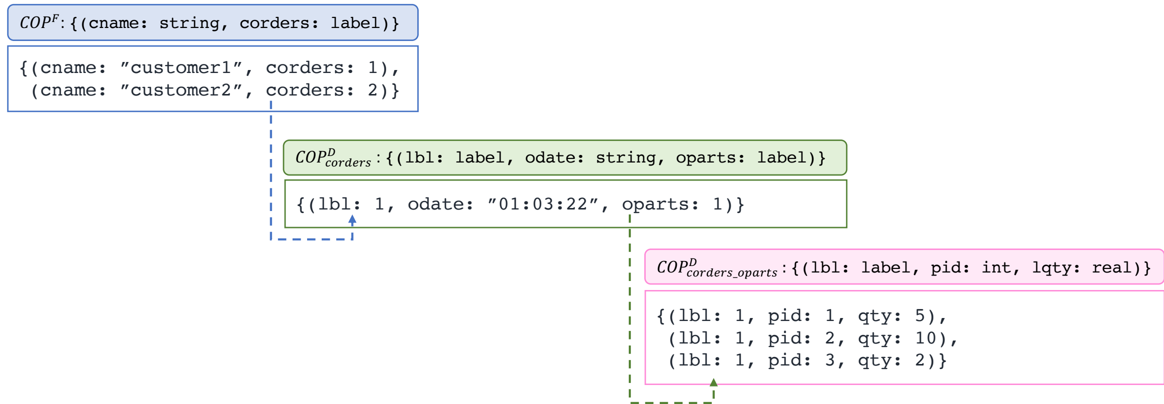


Figure 4.6: The shredded data of COP in relational view with one top-level $\texttt{COP}^{\mathrm{F}}$ and two value dictionaries $\texttt{COP}^{\mathrm{F}}_{\texttt{corders}}$ and $\texttt{COP}^{\mathrm{F}}_{\texttt{corders_oparts}}$

.

$$
\begin{array}{l}
\textsc{TopBag} \Leftarrow \texttt{for } cop^{\mathrm{F}} \texttt{ in } \textsc{MatCOP} \texttt{ union} \\
\qquad\qquad \{\langle \texttt{cname} := cop^{\mathrm{F}}.\texttt{cname}, \texttt{corders} := \texttt{Label}_{\texttt{corders}}(cop^{\mathrm{F}}.\texttt{corders})\}
\end{array}
$$

$$
\begin{array}{l}
\textsc{MatDict}_{\texttt{corders}} \Leftarrow \\
\quad \texttt{for } co^{\mathrm{F}} \texttt{ in } \texttt{MatLookup}\big(\textsc{MatCOP}_{\texttt{corders}}, cop^{\mathrm{F}}.\texttt{corders}\big) \texttt{ union} \\
\qquad \{\langle \texttt{lbl} := co^{\mathrm{F}}.\texttt{lbl}, \texttt{odate} := co^{\mathrm{F}}.\texttt{odate}, \texttt{oparts} := \texttt{Label}_{\texttt{oparts}}(co^{\mathrm{F}}.\texttt{oparts}, co^{\mathrm{F}}.\texttt{odate})\rangle\}
\end{array}
$$

$$
\begin{array}{l}
\textsc{LabDomain}_{\texttt{corders_oparts}} \Leftarrow \\
\quad \texttt{dedup}\big(\texttt{for } x \texttt{ in } \textsc{MatDict}_{\texttt{corders}} \texttt{ union } \{\langle \texttt{lbl} := \texttt{x.oparts}\rangle\}\big)
\end{array}
$$

$$
\begin{array}{l}
\textsc{MatDict}_{\texttt{corders_oparts}} \Leftarrow \\
\quad \texttt{for } l^{\mathrm{F}} \texttt{ in } \textsc{LabDomain}_{\texttt{corders_oparts}} \texttt{ union} \\
\qquad \texttt{let odate} = l^{\mathrm{F}}.co^{\mathrm{F}}.\texttt{odate in} \\
\qquad \texttt{let oparts} = l^{\mathrm{F}}.co^{\mathrm{F}}.\texttt{oparts in} \\
\qquad \texttt{for } op^{\mathrm{F}} \texttt{ in } \texttt{MatLookup}\big(\textsc{MatCOP}_{\texttt{corders_oparts}}, \texttt{oparts}\big) \texttt{ union} \\
\qquad\quad \texttt{for } p^{\mathrm{F}} \texttt{ in } \texttt{Part}^{\mathrm{F}} \texttt{ union} \\
\qquad\qquad \texttt{if } (p^{\mathrm{F}}.\texttt{pid} == op^{\mathrm{F}}.\texttt{pid} \;\&\&\; \texttt{odate} == \text{“}01:03:22\text{”}) \texttt{ then} \\
\qquad\qquad\quad \{\langle \texttt{lbl} := op^{\mathrm{F}}.\texttt{lbl}, \texttt{pname} := p^{\mathrm{F}}.\texttt{pname}, \texttt{total} := op^{\mathrm{F}}.\texttt{qty} * p^{\mathrm{F}}.\texttt{price}\rangle\}
\end{array}
$$

Figure 4.7: The shredded Totals query in relational view with one top-level $\texttt{COP}^{\mathrm{F}}$ and two value dictionaries $\texttt{COP}^{\mathrm{F}}_{\texttt{corders}}$ and $\texttt{COP}^{\mathrm{F}}_{\texttt{corders_oparts}}$

.

avoid any potential impacts on performance with the recursive view, the `BagToDict` construct (Section 4.1) will cast a dictionary in label/bag pair to the relational view. This provides flexibility in the shredded representation, delaying the relational view until the desired stage of compilation. The optimized, materialized shredded query of `Totals` with `BagToDict` applied is provided below. The relational view of `COP` is provided in Figure 4.6.

4.5 Plan Translation

In the plan translation procedure, a lookup operation is a special case of a join between a domain of labels and a materialized expression of dictionary type. Comprehensions first normalize away `MatLookup` operations using the below rules and then the standard join rules from Figure 2.1 are applied.

$$\{e_1 | v \leftarrow \texttt{MatLookup}(p_1, X), \overline{r}, p_2\} = \{e_1 | v \leftarrow X, \overline{r}, if(p_1 == v.lbl)\ then\ p_2\}$$

$$Let\ p_2^{lbl} \ = \ if\ (p_1 == v.lbl)\ then\ p_2$$

$$\{e_1 | v \leftarrow \texttt{MatLookup}(p_1, X), \overline{r}, p_2\} = \{e_1 | v \leftarrow X, \overline{r}, p_2^{lbl}\}$$

From here, U2 and U3 can be applied to produce an (outer)join operation based on the value of u. When u is empty U2 is applied:

$$\texttt{PTrans}(\{e_1 | v \leftarrow X, \overline{r}, p_2^{lbl}\})^{()}_{w}\ E = \texttt{PTrans}(\{e_1 | \overline{r}, p_2[\overline{(w, v)}]\})^{()}_{(\alpha_I^w, \alpha_I^v)}\ (E \bowtie_{p_2^{LLB}[(w,v)]} X)$$

When u is non-empty U3 is applied:

$$\texttt{PTrans}(\{e_1 | v \leftarrow X, \overline{r}, p_2^{lbl}\})^{u}_{w}\ E = \texttt{PTrans}(\{e_1 | \overline{r}, p_2[\overline{(w, v)}]\})^{u}_{(\alpha_I^w, \alpha_I^v)}\ (E ⟕_{p_2^{lbl}[(w,v)]} X)$$

For example, the `MatLookup` operation can be normalized away from the relational view of $\textsc{MatDict}_{\texttt{corders_oparts}}$ in Figure 4.7. This produces a join on the input dictionary and label domain based on the following steps:

$$\{e \ \mid \ l^{\text{F}} \leftarrow \textsc{LabDomain}_{\texttt{corders_oparts}}, op^{\text{F}} \ \leftarrow \texttt{MatLookup}\big(\texttt{MatCOP}_{\texttt{corders_oparts}}, l^{\text{F}}.co^{\text{F}}.\texttt{oparts}\big)\}$$

$$\Downarrow$$

$$\{e \mid l^{\mathrm{F}} \leftarrow \textsc{LabDomain}_{\texttt{corders_oparts}}, op^{\mathrm{F}} \leftarrow \textsc{MatCOP}_{\texttt{corders_oparts}}, op^{\mathrm{F}}.\texttt{lbl} == l^{\mathrm{F}}.co^{\mathrm{F}}.\texttt{oparts}\}$$

$$\Downarrow$$

$$\textsc{MatCOP}_{\texttt{corders_oparts}} \bowtie_{\texttt{lbl}==co^{\mathrm{F}}.\texttt{oparts}} \textsc{LabDomain}_{\texttt{corders_oparts}}$$

Plan translation in the shredded compilation route takes a shredded query and returns a plan for each component, collectively referred to as the *shredded plan.* The shredded compilation route can also produce a plan with or without unshredding in order to return shredded or nested output, respectively. When executed, the shredded plan materializes the output of the shredded query and passes it to the unshredding process to return nested output.

The decision to use plan translation in the shredded compilation route was made to provide ease of integration of the shredded variant into the standard compilation route. The choice for plan translation in the shredded variant is thus the same for that as in standard compilation: it provides an intermediate representation that supports further optimization. An alternative design decision could have been to compile directly from $\mathrm{NRC}_{agg}^{Lbl+\lambda}$ to SQL by providing SQL query strings to SparkSQL; however, this means translating to an external language where additional optimizations can not be applied. The shredded plan in TraNCE is optimized and then described in SparkSQL using the Dataset API through the code generation process. This provides an additional optimization layer that can still benefit from the internal Catalyst optimizer of Spark. Later results will show that such optimizations enable the standard pipeline to outperform SparkSQL.

4.6 Code generation

Dictionaries are represented in Spark using the relational view: a dictionary is a Dataset[T] where T is a Scala case class that contains a `lbl` column. Labels for input value dictionaries are of integer type, meaning `lbl` identifies a column of integer type. For materialized dictionaries `Label` expressions that contain a single encapsulated value are represented as a column of scalar type, ie. $\texttt{Label}_{\texttt{corders}}(cop^{\mathrm{F}}.\texttt{corders})$ is `col("corders")`. Labels that encode multiple values, such as $\texttt{Label}_{\texttt{oparts}}(co^{\mathrm{F}}.\texttt{oparts}, co^{\mathrm{F}}.\texttt{odate})$, are encoded as a case class that encapsu-

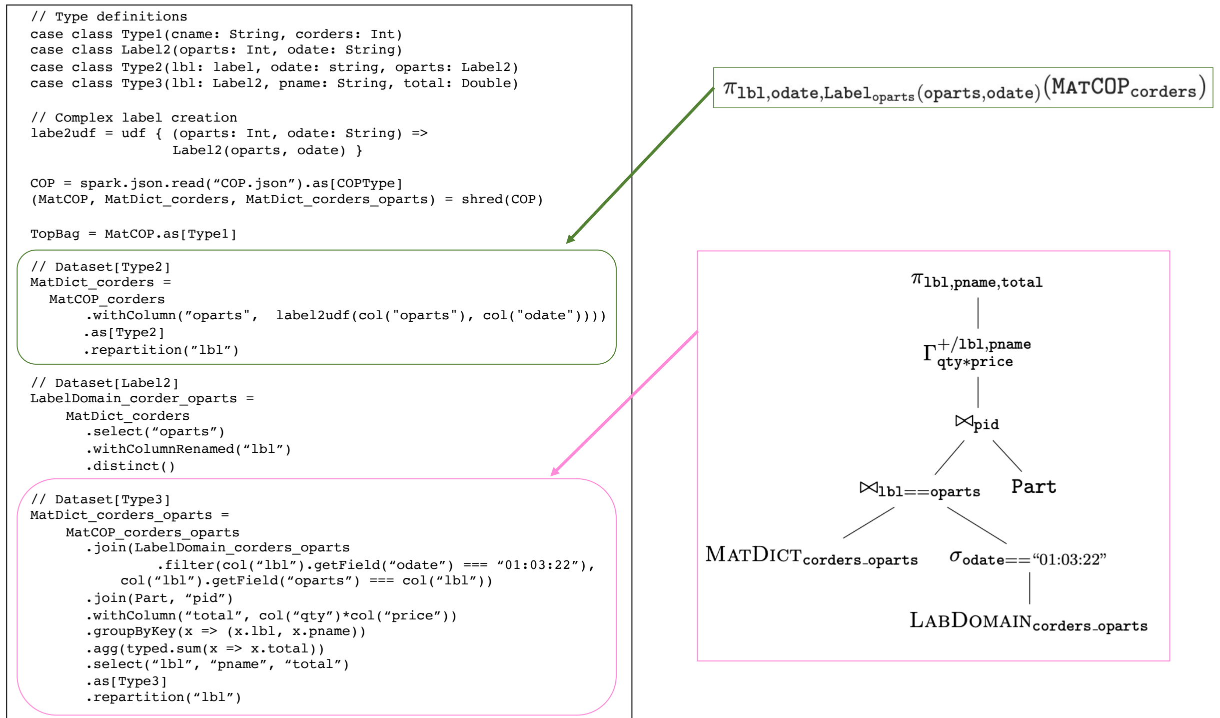


Figure 4.8: The generated Spark program for the shredded `Totals` query following relational view. The plans resulting from the plan translation procedure for both dictionaries are also provided.

late all label components: `case class Label2(oparts: Int, odate: String)`. Figure 4.8 presents the corresponding plans and generated code for the shredded `Totals` query, following the relational view in Section 4.4.

Dictionaries are partitioned by `lbl`, maintaining a *label-based partitioning guarantee* where all values associated to the same label reside on the same partition. This is achieved by repartitioning a dictionary by the `lbl` column. The decision to move all labels to the same partition can be considered counterintuitive since this can still lead to issues that arise from skew. The ability to move all labels to the same partition facilitates the unshredding process, which can introduce joins that trigger shuffle operations. Preemptive repartitioning by label is thus a heuristic that assumes dictionaries are reduced before their final repartition. Any problematic repartitioning that results in data skew is handled by the skew operations discussed in Chapter 5.

Chapter 5

Skew-Handling and Additional Optimizations

The previous chapters presented the standard compilation route, as well as the shredding extensions to support the optimized shredded compilation route. This chapter discusses the additional optimizations that are applied at the later stages in both compilation routes. The plan produced from the comprehension to plan translation procedure is passed to the plan optimizer, where database-style optimizations are applied (Section 5.2). The optimized plan is then passed to the code generation module where additional optimizations are applied (Section 5.3). When the resulting distributed application is executed, dynamic optimizations are applied at runtime to avoid skew-related bottlenecks. The skew-handling methods are outlined in Section 5.1.

5.1 Skew-Resilient Processing

The framework up to this point handles the programming mismatch and the data representation challenges, but has not addressed skew-related bottlenecks. While the shredded representation overcomes skew-related issues due to a small number of top-level tuples and large inner collections, skew-related bottlenecks can still remain due to key-based partitioning strategies. To handle such key-based issues, TraNCE automatically estimates skew-related bottlenecks at runtime and dynamically alters the

query execution strategy to overcome skew for both the standard and shredded compilation routes.

Skew-resilient processing first performs sampling on each data partition to identify heavy keys using a variety of sampling techniques that have not been explored previously. The sampling techniques are discussed in Section 5.1.1. PRPD techniques are then extended to the operators of the plan language, producing skew-aware operations that support plans that adapt to skew. The skew-handling techniques are applied to both the standard and the shredded compilation route, thus the framework supports PRPD techniques that work on both the nested and shredded representation. Section 5.1.2 provides details on the skew-aware plans.

5.1.1 Heavy Key Identification

The core of the skew-handling procedure is based on the identification of heavy keys. If the collected values of a key (key_1) are much larger than the collected values of another key (key_2), then the partition housing key_1 values will take longer to compute than others. Such keys as referred to as heavy, and all others as light. There are four strategies available to identify heavy keys in the skew-handling procedure: full, partial, sample, and slice. Full is the most accurate method, fully identifying heavy keys based on all values across all partitions. Partial identifies heavy keys based on values locally within each partition. Sample identifies heavy keys by randomly sampling a subset (default 10%) of each partition, considering a key heavy when at least a certain threshold of the sampled tuples in a partition are associated with that key.

Since access to data within a partition is via an iterator interface, all of the methods discussed so far require one full iteration over each partition, which can be expensive for large partitions. Slice evaluates heavy keys based on the first range of values (default 1000) in the partition. The sampling percent and slice range are all user-configurable. All methods categorize a key as heavy when the associated values make up a user-specified threshold of the total value (default 2.5%).

The heavy key calculation occurs at runtime prior to the execution of key-based operation. After the heavy keys are calculated they are then cached in a *skew-triple*, which is a data structure that contains: the heavy keys, a bag of values associated to the

light keys, and a bag of values associated to the heavy keys. The set of heavy keys remains associated to a skew-triple until the operator does something to alter the key.

5.1.2 Skew-Aware Plans

The framework automates the skew-handling process with skew-aware plan operators. A skew-aware operator is a variant of a plan language operator (Section 3.2) that accepts and returns a skew-triple. A skew-aware plan contains a light and heavy plan. The *light plan* works on the light component and the *heavy plan* works on the heavy component of the skew-triple. The *light plan* follows the skew-unaware execution for all operators, ensuring the values associated to light keys reside in the same partition. This preserves known partitioning information where possible. The *heavy plan* uses a broadcast-based execution strategy that prevents the movement of associated values to the same node. The heavy values of one input are sent to the heavy values of another input and the computation proceeds locally without shuffling any values. By avoiding the movement of the values associated to heavy keys, the memory footprint of an otherwise memory-intensive task is reduced.

The skew-aware operators assume that the set of heavy keys is small. The threshold used to compute heavy keys puts an upper bound on their number; for example, the threshold of 2.5% means there can be at most $\frac{100}{2.5} = 40$ distinct heavy keys in the sampled tuples per partition. This small domain allows for lightweight broadcasting of heavy keys and, in the case of the skew-aware join operations, the heavy values of the smaller relation.

Figure 5.1 provides the implementation of the main skew-aware operations. All nest operations merge the light and heavy components and follow the standard implementation, returning a skew-triple with an empty heavy component and a null set of heavy keys. Aggregation Γ^{+} mitigates skew-effects by default by reducing the values of all keys. A grouping operation $\Gamma^{\uplus}$ cannot avoid skew-related bottlenecks. More importantly, skew-handling for nested output types would attempt to solve a problem that the shredded representation already handles gracefully.

The encoding of dictionaries as flat bags means that their distribution is skew-resilient by default. In the shredded pipeline, input dictionaries come as skew triples, with

known sets of heavy labels. The skew-aware evaluation of a dictionary involves casting of a bag with the skew-aware `BagToDict` operation. This maintains the skew-resilience of dictionaries by repartitioning only light labels, and leaving the heavy labels in their current location, as shown in Figure 5.1.

Both compilation routes leverage skew-handling methods that maintain proper distribution of values associated to heavy keys. Given that the shredded representation ensures distribution of inner-collections, the shredded compilation method is better-suited to deal with skew-related issues that arise from large nested collections and/or top-level distribution.

The ability to preserve the partitioning guarantee of light keys and ensure the distribution of large collections associated to heavy keys enables skew-resilient processing. This is the basis of the existing PRPD technique, where partial-redistribution occurs on the light keys and partial duplication happens for the heavy keys. This approach will only occur on a join and the light and heavy component will be unioned immediately after execution. The skew-resilience in TraNCE maintains the light and heavy component to consider skew-handling in the context of the whole execution, including nesting, aggregation, and the use of shredded representations; this also leaves opportunities to leverage cached heavy key information.

5.2 Plan Optimizations

The comprehension to plan translation (Section 3.3) may produce suboptimal query plans. The plan is passed to the Optimizer module where database-style optimizations are applied, including pushing down selection, projection, and sum aggregate operators. Unnecessary index operations are also removed as a by-product of the projection-pushing transformation.

Sum aggregates can also be pushed past joins to compute *partial aggregates*, which act locally at each partition and are applied in addition to the original aggregation. The partial sum aggregate operator $\Gamma^{local^{+}/value}_{key}$ performs a key-based sum on each partition without any shuffling. The choice of when to apply a partial aggregate is heuristic-based. Local aggregation is applied when the key is not unique as defined by schema information. It is often the case that nested collections do not have key-related

Plan Operator	Definition

$X \bowtie_{f(x)=g(y)} Y$	`val (X_L, X_H, hk) = X.heavyKeys(f)`
	`val Y_L = Y.filter(y => !hk(g(y)))`
	`val Y_H = Y.filter(y => hk(g(y)))`
	
	`val light = X_L.join(Y_L, f === g)`
	`val heavy = X_H.join(`
	`  Y_H.hint("broadcast"), f === g)`
	
	`(light, heavy, hk)`

$\Gamma^{agg\,value}_{key} X$	`val unioned = X_L.union(X_H)`
	
	`// proceed with light plan`
	`val light = ...`
	
	`(light, sc.emptyDataset, null)`

BagToDict X	`val (X_L, X_H, hk) =`
	`  X.heavyKeys(x => x.label)`
	
	`val light = X_L.repartition(x => x.label)`
	`val heavy = X_H`
	
	`(light, heavy, hk)`

Figure 5.1: Skew-aware implementation for the plan language operators using Spark Datasets.

schema information; thus, unnest and outer-unnest operations are always subject to partial aggregation. Since these operators can cause data blow ups, it is assumed that applying the local aggregation prior to flattening will lead some data reduction in the unnested result.

The *aggregation pushing* algorithm is responsible for applying the partial aggregations to the operators in an input plan when the heuristic is satisfied. Aggregation pushing is a recursive transformation that works top-down over a query plan. Figure 5.2 presents the transformation steps based on the root plan operator for any given subplan, assuming the heuristic is satisfied. The aggregation pushing function $\mathcal{L}^v_k(e)$ takes an input plan e, a key attribute set k, and a value attribute set v. The attributes sets are initially empty, but are populated by the *key* and *value* parameters of the sum

	Plan Operator	Heuristic-Based Transformation
1	$\Gamma^{+/value}_{key}(a)$	$\Gamma^{+/value}_{key}(\mathcal{L}^{v \,\cup\, value}_{k \,\cup\, key}(e))$
2	$\Gamma^{\uplus/value}_{key}(e)$	$\Gamma^{\uplus/vs}_{ks}(\mathcal{L}^{v}_{k}(e))$
3	$\pi_{a_1,\ldots,a_k}(e)$	$\pi_{a_1,\ldots,a_k}(\mathcal{L}^{v}_{k}(e))$
4	$e_1 ⟕_a e_2$	$\mathcal{L}^{k \backslash \alpha^{e_2}}_{v \backslash \alpha^{e_2}}(e_1) ⟕_a \mathcal{L}^{k \backslash \alpha^{e_1}}_{v \backslash \alpha^{e_1}}(e_2)$
5	$e_1 \bowtie_a e_2$	$\mathcal{L}^{k \backslash \alpha^{e_2}}_{v \backslash \alpha^{e_2}}(e_1) \bowtie_a \mathcal{L}^{k \backslash \alpha^{e_1}}_{v \backslash \alpha^{e_1}}(e_2)$
6	$\not\mu^{a}\ (e)$	$\Gamma^{local+/v}_{k}(\not\mu^{a}\ (\mathcal{L}^{k}_{v}(e)))$
7	$\mu^{a}(e)$	$\Gamma^{local+/v}_{k}(\mu^{a}(\mathcal{L}^{k}_{v}(e)))$
8	$\sigma_p(e)$	$\Gamma^{local+/v}_{k}(\sigma_p(e))$

Figure 5.2: Local aggregation optimization

aggregate operator (line 1). The transformation passes through projections and nest operations (lines 2-3). Joins and outer-joins will only pass attributes in k and v that are in the scope of the left and right subplans; denoted as set difference between the attributes of the subplan α^e (line 4 and 5). The partial aggregation operator is appended to the unnest or outer-unnest operator, using the k and v sets up to that point as the respective parameter values (line 6-7). During code generation, the partial aggregation is merged with the application of the unnest operation to avoid flattening before reducing. The local aggregation is applied to a selection operator only when k, collectively, is not a key (line 8). The case of a leaf node - input or materialized data source - follows the selection operator when p is equal to true.

Figure 5.3 shows the optimized plan for the `Totals` running example, with unnecessary indices removed, selections pushed, and local aggregations applied. The final outer-unnest has partial aggregation applied, computing the partial sum of `qty` grouped by $\{$`copID`, `coID`, `cname`, `odate`, `pname`$\}$; the intermediate value is denoted `total`. The partial aggregate is taken locally, and after joining with `Part` the final sum globally aggregates the values.

Local aggregation is performed by iterating over each partition (`it`) in a Dataset (`df`) and applying the local sum for each key. Specifically, `foldLeft` operates on a partition

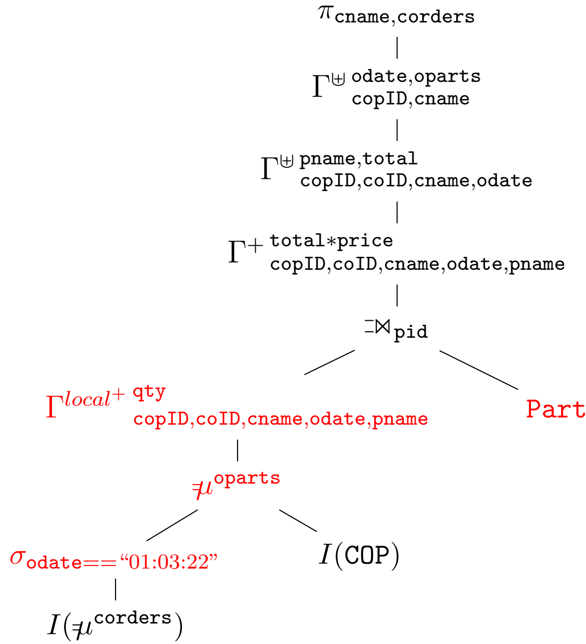


Figure 5.3: The optimized query plan for the `Totals`running example. Index operator removal, selections pushed, and local aggregation optimizations are colored red.

iterator, which updates a Scala HashMap with every value. For example, for a variable v with key attribute *key* of String type and *value* attribute *value* of Double type, the local aggregation is defined as:

```
df.mapPartitions( it =>
  it.foldLeft(HashMap[String, Double].withDefaultValue(0.0))(
    (acc, v) => {acc(v.key) += v.value; acc})).iterator
)
```

The runtime of local aggregation is determined by the time to iterate over the partition and build up the intermediate object. Partial aggregation is only beneficial when there is some guarantee that performing the operation will reduce the data enough to outweigh these operations. The use of schema information for local aggregation application is thus not always a win. For example, `oparts` could be small (contains only a few tuples) so partial aggregation brings little benefit. Alternatively, `oparts` could contain only distinct `pid` attributes, which would not be identified without nested schema information. The benefit of performing local aggregation under these conditions would only be measurable with catalog statistics. This measure would need to

determine if the cost of performing the local aggregation is amortized to the amount of reduction, ie. has a high *reduction factor*. A reduction factor is the ratio of distinct values to the number of rows in the candidate dataset. To calculate the reduction factor both column and input level statistics are required. This is the focus of future and ongoing cost-based optimization work.

5.3 Code Generation Optimizations

The optimized plan is passed to code generation where additional optimizations are applied. As mentioned in Section 5.2, partial aggregations are merged with unnest and outer-unnest operators. Nest operations are also merged with join operations. These optimizations collectively eliminate unnecessary data as early as possible and thus reduce data transfer during distributed execution. This section provides the code for these optimizations.

5.3.1 Partial Aggregation and Unnest Merge

When a partial aggregate operation follows immediately after an (outer)unnest operation then the two are merged. The goal is to reduce the flattened result as much as possible. If the local aggregation operated on the output of the unnest operation, as described by the plan, the local aggregation is less likely to benefit since the unnest could blow up the input size. The local aggregation follows from the implementation in Section 5.2, but applies the local aggregation to each of the inner nested collections within a flat-map to return a flattened and aggregated result tuple. For example, the subplan:

$$\Gamma^{local+\ \mathtt{qty}}_{\mathtt{copID,coID,cname,odate,pname}}\ (\mu^{\mathtt{oparts}}(\mathtt{DF}))$$

will produce the following generated code:

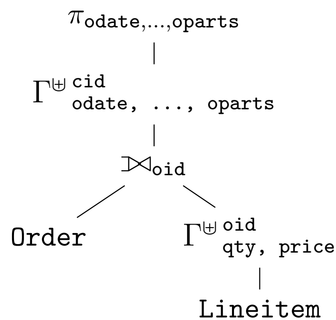
```
DF.flatMap( x =>
    x.oparts.foldLeft(HashMap[KT, Double].withDefaultValue(0.0))(
        (acc, op) => {
          acc((x.copID, x.coID, x.cname, x.odate, Some(op.pname)) +=
              op.qty; acc
        })).iterator)
```

The type of the key, represented as `KT`, is `(Long, Long, String, String, String)`. The `flatMap` transformation operates on the `oparts` collections, building a HashMap for each of the top-level tuples x. The key is constructed from both top-level and lower-level attributes, summing over the product of the value attributes. The merging of the local aggregation and the unnest operators has performed local aggregation and flattening at the same time, thus reducing the amount of data returned from the unnest operation.

5.3.2 Nest and Join Merge

A join followed by a nest operation is directly translated into a cogroup, which is a Spark primitive that leverages logical grouping to avoid separate grouping and join operations. This is beneficial when building up nested objects with large input bags. The cogroup is implemented in a left-outer fashion, persisting empty bags from the right relation for every matching tuple in the left relation.

Consider a plan that performs a join and a nest on `Order` and `Lineitem`:



This will be translated to the following during code generation:

```
Orders.groupByKey(o => o.oid)
      .cogroup(Lineitem.groupByKey(l => l.oid))(
         case (key, orders, lineitems) =>
           val oparts =
             lineitems.map(l => (l.pid, l.lqty)).toSeq
           orders.map(o => (o.odate, oparts)))
```

First, both `Order` and `Lineitem` are logically grouped by `oid`, and their results are cogrouped. The cogroup operation then maps over the associated collection of `Order`

tuples to pair each with the associated collection of `Lineitem` tuples. This operation has thus merged the join and grouping of these relations into one step. Since the join and nest operations are both shuffle inducing, the merging of these operators results in only one shuffling stage instead of two.

Chapter 6

TraNCE Architecture and Additional Features

This chapter presents the TraNCE architecture that consolidates all the components discussed thus far into a compilation framework. The architecture of TraNCE is presented in Figure 6.1. An NRC_{agg} query is submitted to the framework and is compiled using the standard compilation route (Chapter 3) or the shredded compilation route (Chapter 4). The standard compilation route uses the plan translation procedure to produce a query plan from an NRC_{agg} program. The plan is then optimized and passed to the code generation module, which produces a distributed application defined in the SparkSQL Dataset API. The resulting distributed application can then be executed on top of Spark to return materialized results with nested output type.

The shredded compilation route is implemented as an extension on the standard compilation route. The input NRC_{agg} program is passed to the shredding module which produces the materialized, shredded query. Since the materialization procedure guarantees that only `Label` and `MatLookup` expressions remain in the shredded query, minimal extensions to the standard route are required. The shredded query is passed to plan translation and the shredded plan results. The shredded plan is optimized and passed to the code generation module, producing a SparkSQL application as in the standard route. Value unshredding is included in the application to return the materialized results with nested output type. The shredded compilation route occurs com-

pletely in the background such that the user never interacts with the shredded representation directly.

The rest of this chapter presents a few additional features of the framework. First, the assumption and process around input data definitions are discussed (Section 6.1). To facilitate any overhead with constructing an NRC_{agg} query, a frontend interface is provided that supports interactive query building. The interface also provides exploring the output of compilation stages and viewing results. The motivation and walk-through of the frontend interface are described further in Section 6.2. Distributed applications can be compiled as an executable or as an Apache Zeppelin notebook. Notebooks provide further support for UDFs to interface with external libraries. UDF support is discussed in Section 6.3, including UDF support in the shredded compilation route and optimization with hints.

6.1 Input Definitions and Sanitation

A TraNCE data input is required to have a schema definition that follows the nested data model. There is no need for a user to define the associated shredded type since this will be automatically handled in the shredding module. The schema definition is accompanied with a Spark loader definition that conforms to the schema.

Many datasets today come in formats that are richer than the nested model, such as JSON. When processing richer formats in TraNCE, the data needs to be sanitized in order to account for things like missing attributes. The system assumes data sanitation is handled in the Spark data loader. For example, a tuple from the `Part` dataset that is missing a `price` field: $\langle \texttt{pid} : 7, \texttt{pname} : \text{“}\mathit{Part}7\text{”}\rangle$. To account for the missing attribute, a pass over the `Part` dataset is required to replace all missing values with a user-preferred default, such as 0.0 for missing *real* types: $\langle \texttt{pid} : 7, \texttt{pname} : \text{“}\mathit{Part}7\text{”}, \texttt{price} : 0.0\rangle$. Future work should consider the performance impacts of sanitation and how a richer model could avoid sanitation altogether.

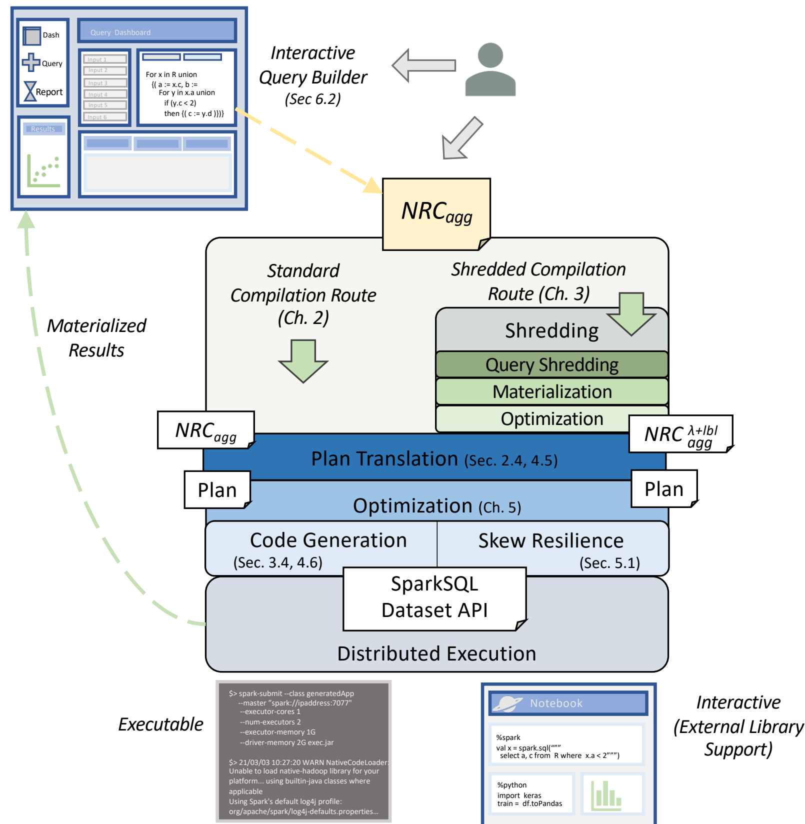


Figure 6.1: System architecture.

6.2 Interactive Frontend

This section provides an overview of the web-based TraNCE frontend. The interface supports interactive query building, exploring compilation outputs, viewing nested results, and browsing execution metrics. One of the goals of the interactive interface is to ease the process of restructuring complex data sources and evaluating datasets at large-scales. Another goal is to interface with a variety of external software and statistical libraries in an interactive environment. A brief walk-through of the interface components is provided next.

Interface components and usage. The query builder view of TraNCE (Figure 6.2) allows users to build queries without in-depth knowledge of NRC. The builder uses blockly [65] to provide NRC_{agg} expression "blocks", which can be pieced

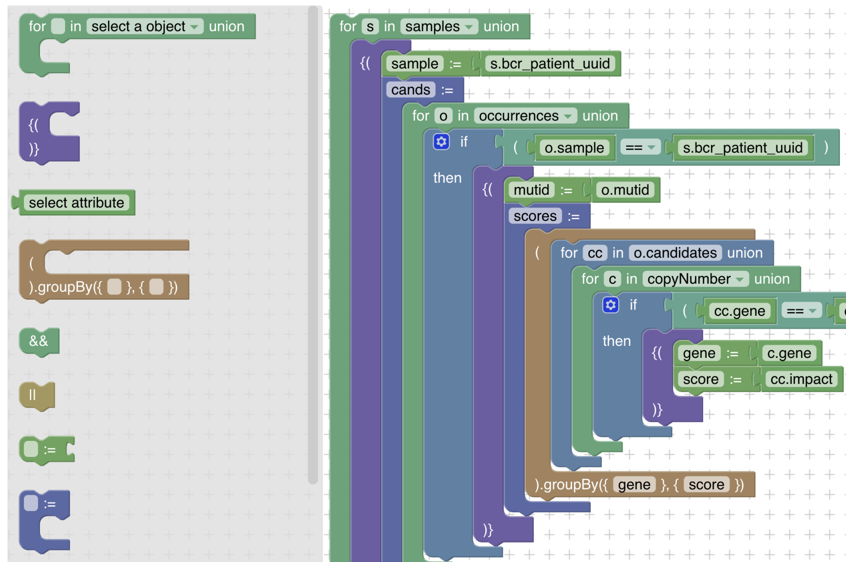


Figure 6.2: Query builder view

Figure 6.3: Compilation view

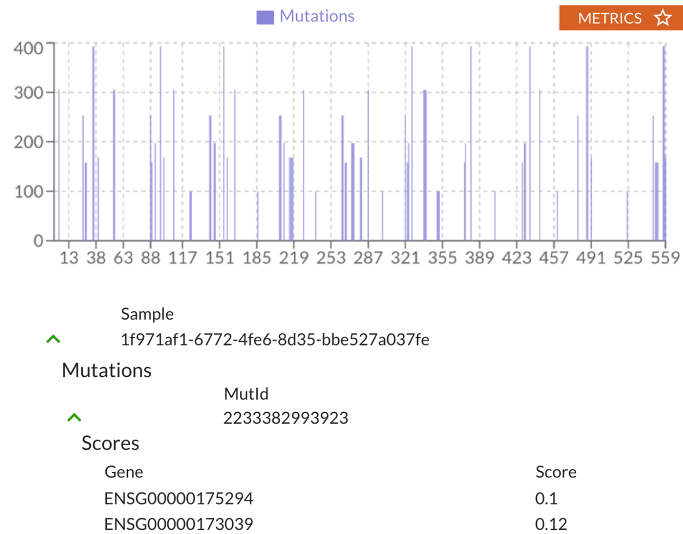


Figure 6.4: Results view

together to form a TraNCE program. Inputs are selected within relevant blocks providing the scope for subexpressions. Queries can be imported or deleted using the top right-hand corner menu. An additional dashboard view provides a comprehensive list of all queries; the user can edit, execute, or delete queries from this list.

The compilation view provides an exploratory interface for the outputs of the TraNCE compilation process. When compilation is triggered, the compilation window shows the shredded NRC query and the plan produced by the shredded compilation route; the user can also see the standard plan produced without shredding. Mousing over the syntax trees allows the user to see the links between the NRC source and the artifacts produced by compilation. Figure 6.3 shows an example shredded query and plan that are the outputs from the compilation phase.

The compilation view also allows the user to see the impact of changes in the materialization and shredding strategy. After compilation, the interface gives options for execution, including moving to a notebook that enables interactive and external analysis. Users can execute programs and will be notified of completion while they continue building queries and exploring compilation outputs.

The results view (Figure 6.4) displays the raw output of an execution. The user can

browse through the data to drill down to a new level, and a bar graph provides an overview of a given inner collection within the output. Runtime metrics can also be explored through the interface. The distribution of partition sizes for the longest running task is provided, which gives insight into skew and its management within the TraNCE architecture. This view also has links to statistics produced by the target parallel processing framework.

6.3 User-Defined Functions

User-defined functions are available for use in NRC_{agg} via the `udf` expression. To support UDFs, the system must be aware of their function names and definitions. This section describes the additional components required for supporting user-defined functions and associated metadata in TraNCE. Section 6.3.1 outlines how UDFs with internal types, ie. the types in Figure 3.1, are registered and used within a NRC_{agg} program. Section 6.3.2 discusses the extensions to the type system required to support external types. The remaining sections discuss how UDFs are integrated into the shredded compilation route and further optimized with hints.

6.3.1 Compilation Extensions

TraNCE manages user-defined functions with a UDF-registry, where users will first define a UDF before it can be used within a NRC_{agg} program. The definition of an internally-typed UDF includes the name of the udf, the output type of the udf, and a corresponding `.udf` file containing the function definition in Scala.

Recall the UDF expression from the NRC_{agg} source language, $\texttt{udf}^{name}_{params}(e)$ presented in Section 3.1. The *name* attribute matches the registered name of the udf. The *params* attribute list is provided as part of the function definition, as well as the expected input type of *e*.

UDF compilation is handled at each module within the framework. From the view of the compiler, the registered output type of the `udf` is the most important attribute since this enables type-checking. All other attributes are passed through to code generation, where the UDF is translated into the respective function call with proper

placement of the provided parameters. The code generation process keeps track of the UDFs that are referenced in the input program and accesses their function definitions from the UDF registry, including them in the generated Spark/Scala application. The following two examples overview the handling of UDFs within TraNCE. Example 5 presents a primitive-typed UDF.

Example 5. This example presents how a primitive function can be used in an NRC_{agg} program. Consider a UDF `exampleudf` that appends a string to the value of a tuple attribute; the appended string is a user-specified parameter and the tuple attribute is the input. The UDF is first registered for use in the framework with the name `exampleudf` and *string* output type. The definition is provided in a file named `exampleudf.udf`, containing the function definition as defined in the Spark/Scala API:

```
def exampleudf(input: Column, add: String): Column = concat(x, lit(add))
```

Once registered, the function can then be used in a TraNCE program. For example, a user can append the string "ID" to each of the `cname` attributes in the `COP` dataset:

$$\begin{array}{l} \texttt{CustomerIDs} \Leftarrow \\ \texttt{for } cop \texttt{ in COP union} \\ \quad \{\langle \texttt{cname} := \texttt{udf}^{\texttt{exampleudf}}_{''ID''}(cop.\texttt{cname})\rangle\} \end{array}$$

Recall that the compilation framework will translate source NRC_{agg} to a form designed for execution in bulk, ie. plan language or Spark code. This means that the UDF definition must follow the bulk execution strategy of the TraNCE plan language, as shown in the Spark/Scala definition of `exampleudf`; whereas, the associated $\texttt{udf}^{\texttt{exampleudf}}$ expression is applied within a NRC_{agg} program as a *string* $\Rightarrow$ *string* transformation. This design is a by-product of passing a `udf` expression through a compilation route that targets bulk, distributed collection APIs.

6.3.2 External Types

Data science workflows often require the use of statistical libraries from external languages, such as Python. TraNCE thus needs to handle external types to support UDFs. This is achieved using *context switching*. Context switching is a feature provided in notebook frameworks that allows passing data representations between code interpreters - such as Scala to Python; thus, externally typed UDFs can only

be used when compiling out to a Zeppelin notebook. To support such a pipeline, TraNCE allows users to register a UDF with an external type. The system then provides native internal-to-external UDFs that cast an external type back to an internal type.

6.3.3 Shredding UDFs

This section discusses the relevant extensions required to support user-defined functions in the shredded compilation route. UDFs need to be handled in both the query shredding and materialization steps of the shredding transformation. Given that the goal of query shredding is to produce a set of flat queries that operate on flat inputs, then shredding a UDF should produce a set of UDFs with flat output types that operate on a set of flat input types. The shredded UDF thus consists of a collection of UDFs that each has a flat output type, with the input to each shredded UDF in the collection is the full set of shredded inputs.

The query shredding rule for UDFs is provided in line 15 of Figure 4.4; *name* and *params* are passed through the transformation and are thus emitted in the figure. For each `udf` expression, the query shredding algorithm expects an expression $\mathcal{F}(\texttt{udf})$ and $\mathcal{D}(\texttt{udf})$. For primitive-typed UDFs, $\mathcal{F}(\texttt{udf})$ is merely the originally defined UDF and $\mathcal{D}(\texttt{udf})$ is an empty dictionary. For bag-typed UDFs, the structure of `udf` within the compiler cannot be analyzed so these functions must be supplied by the user. Since the user cannot provide symbolic expressions in the symbolic shredding phase, $\mathcal{F}(\texttt{udf})$ and $\mathcal{D}(\texttt{udf})$ are left as stubs. The `ShredUdf` stub is provided as an extended dictionary type that takes the symbolic representation of the shredded input, the associated *params* and *name* attributes; thus, `ShredUdf` is a dictionary type that passes the UDF metadata captured in the source `udf` expression through to the materialization phase.

At the materialization stage, the shredded output type - stored in the UDF registry - is used to produce a set of materialized UDFs, each of the flat type expected from the materialization algorithm. The user is responsible for providing the materialized `udf`, including the the top-level and the flat component associated to each level of $\mathcal{D}(\texttt{udf})$. These are registered as shredded function definitions in the UDF registry. Example 6 describes the process of supporting a UDF within the shredded compilation route.

Example 6. Consider a UDF `copIdentity` that performs the identity function on the

`COP` data source and thus returns a bag type. This UDF is registered as `copIdentity` with the following definitions:

```
def copIdentity(input: Dataset[COPType]): Dataset[COPType] = input
```

The function can then be used in an NRC_{agg} program:

$\texttt{COPIdent} \Leftarrow \texttt{udf}^{\texttt{copIdentity}}(\texttt{COP})$

Since the `COP` data source has a top-level and two nested levels, the shredded definition of `copIdentity` will consist of a top-level function and two additional functions producing the flat representations of the two nested levels; each of these functions takes all three flat collections of the shredded `COP` data as input. The function definitions in Spark/Scala for each UDF in the shredded UDF collection for `copIdentity` are below. Since this function performs the identity, each function returns the associated shredded input for that level:

```
def MatcopIdentity(top: Dataset[Type1], first: Dataset[Type2], second:
    Dataset[Type3]): Dataset[Type1] = top

def MatcopIdentity_corders(top: Dataset[Type1], first: Dataset[Type2],
    second: Dataset[Type3]): Dataset[Type2] = first

def MatcopIdentity_corders_oparts(top: Dataset[Type1], first:
    Dataset[Type2], second: Dataset[Type3]): Dataset[Type3] = second
```

where the case class types, ie. `Type1`, are the case classes defined from Figure 4.8.

The NRC_{agg} program can then be compiled using the shredded compilation route. The query shredding transformation will make a recursive call shredding the variable reference of `COP` to return $\mathcal{F}(\texttt{COP})$ and $\mathcal{D}(\texttt{COP})$. These are wrapped in the `ShredUdf` stub along with the `udf` name `copIdentity` and passed to the materialization phase. Again, since there is a top-level and two nested collections in the type of `COP`, the final, materialized, output of the shredding transformation will have three materialized `udf` expressions:

$\textsc{MatCOPIdent} \Leftarrow$
$\texttt{udf}^{\textsc{MatcopIdentity}}(\textsc{MatCOP}, \textsc{MatCOP}_{\texttt{corders}}, \textsc{MatCOP}_{\texttt{corders_oparts}})$

$\textsc{MatCOPIdent}_{corders} \Leftarrow$
$\texttt{udf}^{\textsc{MatoccurIdentity}_{cands}}(\textsc{MatCOP}, \textsc{MatCOP}_{\texttt{corders}}, \textsc{MatCOP}_{\texttt{corders_oparts}})$

$\texttt{MatCOPIdent}_{\texttt{corders_oparts}} \Leftarrow$
$\texttt{udf}^{\texttt{MatOccurIdentity}_{\texttt{candidates_consequences}}}(\texttt{MatCOP}, \texttt{MatCOP}_{\texttt{corders}}, \texttt{MatCOP}_{\texttt{corders_oparts}})$

After the shredding transformation, the compilation route proceeds as in the standard route, following through to code generation where the shredded definitions and corresponding function calls are applied in the Spark application.

6.3.4 UDF Hints

User-defined functions can be optimized with the help of hints, as first discussed in Section 2.9. A hint parameter can be provided within a UDF in TraNCE to open up optimization opportunities in the Optimizer module. Current UDF hint support is designed for a filter-specific case study that is discussed in Chapter 9. This section outlines how hints are handled in the framework.

Hints are made available to the framework via the UDF registration process, where it can then be provided as a parameter in a `udf` expression. The `udf` passes through the stages of compilation, where the hint is translated to a $\texttt{udf}^{hint}$ plan operator, denoting a selection on the input to the original `udf`. The hint operator is injected into the query plan before the original `udf` call. The hint parameter is then removed from the original `udf` node and the optimized query plan is sent to code generation. The following example overviews the handling of a UDF hint within TraNCE.

Example 7. Consider a function `trainudf` that performs feature selection and trains a model on an input feature matrix. This UDF will be used in an analysis that predicts customer nation given the total amount spent on each part. The program corresponding to this analysis is defined as follows:

```
Features ⇐
 for n in Nation union
  {⟨nname := n.nname, features :=
   sumBy^{total}_{cname,pname}(
    for c in Totals union
     for o in c.corders union
      for p in o.oparts union
       {⟨cname := c.cname, pname := p.pname, total := p.total⟩})
```

$\texttt{Accuracy} \Leftarrow \texttt{udf}^{\texttt{trainudf}}(\texttt{Features})$

The feature selection process in trainudf will only keep parts that are found to correlate with nation. A hint can be used to speed up feature selection by filtering out low-valued parts that are not likely to be selected as a feature. The threshold hint achieves this by removing all values below a threshold of 0.01. The hint is registered with the following Scala function definition:

```
def threshold[T](input: Dataset[T]): Dataset[T] =
  input.filter(col("total") > 0.01)
```

where T is the type of the input Dataset. Once registered the threshold hint can then be called within the udf parameters of the program:

$\texttt{Accuracy} \Leftarrow \texttt{udf}^{\texttt{trainudf}}(\texttt{Features}, \texttt{threshold}).$

When the above program is compiled, the optimizer will recognize the hint parameter in the udf call. The $\texttt{udf}^{\texttt{threshold}}$ operator is injected above the original udf node to produce the following optimized plan:

```
udf^trainudf
     |
udf^threshold
     |
  Features
```

The plan is passed to code generation and an application is produced that applies the threshold function to Features prior to calling trainudf. The threshold filter removes the feature scores that are below 0.01 before executing the trainudf function, which should reduce the workload for the downstream UDF call.

Hints are currently based on a filter-specific case study (Chapter 9). Future work should consider how hints are represented in a query plan for further optimization. For example, the injection of the operator prior to the original udf call is specific to hints that filter an input before a downstream function call. For simple filters, the corresponding selection operator could be pushed further with the database-style selection pushing. Complex filters, however, will require more advanced analysis to determine what can be pushed further.

Chapter 7

Performance Comparison of Nested Data Systems

This chapter assess current approaches for large-scale processing of nested data, including manual flattening, document stores, and object-relational DBMS extensions. The performance of these systems was compared to both compilation routes of the TraNCE framework. A novel nested TPC-H benchmark is used for the evaluation and is available at [30]. To the best of our knowledge, this is the only nested data benchmark that supports increasing nested depth, increasing size of inner collections, and increasing amounts of skew. The benchmark provides a stress-test ranging from low, moderate, and extreme scenarios for processing nested data in a distributed environment. Queries with both flat and nested input and output types are considered. The chapter first presents the query benchmark, discusses the experimental setup, and then evaluates performance of all systems. The evaluations are organized by query type: flat-to-nested, nested-to-nested, and nested-to-flat. Skewed datasets are explored separately from non-skewed. A summary of the experiments, including the questions asked is provided below.

Flat-to-nested queries with non-skewed data. The flat-to-nested queries on non-skewed data build nested collections from flat inputs. The following questions are asked:

- How do top-level distribution strategies perform as query depth increases in the out-

put? The size of inner collections and the number of attributes in the response are also considered.
- What are the side-effects of unshredding the shredded representation when all inputs are flat? This looks at the join overheads and data shuffling costs that are a consequence of unshredding.

The results affirm that large-sized inner collections and top-level distribution strategies are failing points for current systems. These systems crash due to memory saturation when building up a nested collection with large inner collections and small numbers of top-level tuples. The shredded representation is the only survivor that scales to deeper levels of nesting in the output.

Nested-to-nested queries with non-skewed data. The nested-to-nested queries on non-skewed data look at operating on nested inputs, applying operations at deeper levels of nesting, and returning nested outputs. There are two areas of focus for this experiment:

- How do top-level distribution strategies perform as query depth increases in both the input and the output? The size of inner collections and the number of attributes in the response are also considered.
- Nested joins and aggregations: what is the impact of performing joins and aggregations at non-root levels? A comparison of the flattening methods of current systems to the shredded representation is provided.

The results show that this is a worst case for current systems, which crash from memory saturation when regrouping the flattened result. The shredded representation scales to deeper levels of nesting in both input and output. Operations are applied locally to the succinct representation that leads to more light-weight aggregation and data reduction that reduces unshredding costs.

Nested-to-flat queries with non-skewed data. The nested-to-flat queries on non-skewed data operate over multiple levels of nested input to produce flat output. There are two areas of focus for this experiment:

- How do all systems behave when operating on nested input to return flat output as query depth increases in the input? The size of inner collections and the number of attributes in the response are also considered.

- What is the overhead to reassociate levels in the shredded representation to return flat output? This looks at the join overheads and data shuffling costs.

When data can be properly distributed, current systems are less burdened when processing nested collections with flat output than with nested output. However, the shredded representation presents more opportunities for localized aggregation that supports scaling to deeper levels of nesting in the input.

Nested-to-nested queries with skewed data. The SQL interface to SparkSQL and the TraNCE compilation routes are evaluated for increasing amounts of skew in nested input. The following questions are asked:

- What is the overhead in sampling heavy keys for non-skewed data?
- How do the systems behave as the amount of data skew increases in inner collections with and without skew-handling methods?
- What added skew-handling benefits are observed with the shredded representation?

The results show that while there is an overhead with heavy key sampling, this is amortized in the presence of data skew. The systems that use flattening methods (SparkSQL and the standard compilation route) can only survive to moderate amounts of skew regardless of skew-handling techniques. The shredded compilation route survives all amounts of skew, showing natural skew-resilience in the shredded representation and exhibiting a performance increase when used in combination with skew-handling techniques.

7.1 Our Nested TPC-H Benchmark

Our nested TPC-H benchmark is based off the standard TPC-H schema [66]. The benchmark includes a suite of five flat-to-nested, five nested-to-nested, and five nested-to-flat queries - all with 0 to 4 levels of nesting and organized such that the number of top-level tuples decrease as the level of nesting increases. All queries start with Lineitem table at level 0, then group at every level across Orders, Customer, Nation, and Region, as the level of nesting increases. Each query has a *wide* variant that keeps all attributes and a *narrow* variant that follows the grouping with a projection at each level. At the higher levels, the narrow schema keeps only a single attribute, e.g. `odate`

for Orders, `cname` for Customer, etc. The wide variant returns all attributes except at the lowest level, which still just has the `pid` and `qty` of Lineitem.

At each level, an increasing amount of data is grouped into the nested collections reflecting the relevant stress on a distributed framework. The stress levels can be organized as follows: 0 low stress, 1-2 moderate stress, and 3-4 high stress. The wide variants add additional stress on the systems in comparison to the narrow variants. The term *stress-test* is used to reflect the impact of increasing stress on the system.

Flat-to-nested queries group relational inputs to return nested output. *Nested-to-nested queries* take the materialized output of the flat-to-nested queries as input, applying a nested join and nested aggregation at the lowest level, and returning nested output. *Nested-to-flat queries* take the materialized output of the flat-to-nested queries as input, applying a nested join at the lowest level and an aggregation at the top level to return flat output. The NRC_{agg} program and the optimal plan produced from both the standard and shredded compilation routes. Where relevant, the plan corresponding to the unshredding process is provided. The query with the highest amount of nesting is provided for each category, since queries with fewer levels of nesting are merely subsets of this query.

7.1.1 Flat-to-nested

The five flat-to-nested queries perform iterative grouping of relational inputs to return nested output. This starts with the Lineitem table (0 levels), Lineitem grouped by Orders (`oparts`), `oparts` grouped by Customers (`corders`), `corders` grouped by Nation (`ncusts`), `ncusts` grouped by Region (`rnations`). The lowest level persists the Part key `pid` and the quantity of Lineitem `qty`. For scale factor 100, query results have 600 million, 150 million, 15 million, 25, and 5 top-level tuples.

Figure 7.1 presents the NRC_{agg} program with four levels along with the standard and shredded compilation plans. The sequential join-nest operations in the standard plan will be merged into cogroups during code generation as described in Section 3.4. The shredded plan is prior to unshredding. For flat-to-nested queries, the unshredding plan is identical to the plan produced by the standard compilation route, with each input relation represented as a top-level bag.

```
RNCOP ⇐
for r in Region union
 {⟨rname := r.rname, ..., rnations :=
  for n in Nation union
   if r.rid == n.rid then
    {⟨nname := n.nname, ..., ncusts :=
     for c in Customer union
      if n.nid == c.nid then
       {⟨cname := c.cname, ..., corders :=
        for o in Order union
         if c.cid == o.cid then
          {⟨odate := o.odate, ..., oparts :=
           for l in Lineitem union
            if o.oid == l.oid then
             {⟨pid := l.pid, ..., qty := l.qty⟩}⟩}⟩}⟩}⟩}
```

(a) NRC_{agg} program

(b) Standard compilation plan

$$\texttt{RNCOP}_{Top} := \pi_{\texttt{rname},\ldots,\texttt{rnations:=rid}}(\texttt{Region})$$
$$\texttt{rnations}_{Dict} := \pi_{\texttt{lbl:=rid},\texttt{nname},\ldots,\texttt{ncusts:=nid}}(\texttt{Nation})$$
$$\texttt{ncusts}_{Dict} := \pi_{\texttt{lbl:=nid},\texttt{cname},\ldots,\texttt{corders:=cid}}(\texttt{Customer})$$
$$\texttt{corders}_{Dict} := \pi_{\texttt{lbl:=cid},\texttt{odate},\ldots,\texttt{oparts:=oid}}(\texttt{Order})$$
$$\texttt{oparts}_{Dict} := \pi_{\texttt{lbl:=oid},\texttt{pid},\texttt{qty}}(\texttt{Lineitem})$$

(c) Shredded compilation route plan

Figure 7.1: Flat-to-nested query with four levels of nesting, from which queries with zero to three levels of nesting can be derived. Ellipses represent the additional fields that may be present based on narrow and wide schemas.

7.1.2 Nested-to-nested

The five nested-to-nested queries operate on the materialized result of the flat-to-nested queries, producing the same hierarchy and number of top-level tuples in the nested output. The queries include a join with `Part` followed by a sum aggregate $\texttt{sumBy}_{\texttt{pname}}^{\texttt{qty}\times\texttt{price}}$ at the lowest level. Figure 7.2 presents the NRC_{agg} program as well as the corresponding standard and shredded query plans for four levels of nesting.

7.1.3 Nested-to-flat

The five nested-to-flat queries follow the same construction as the nested-to-nested queries, but apply $\texttt{sumBy}_{name}^{\texttt{qty}\times\texttt{price}}$ at top-level, where *name* is the top-level attributes for the given level. These queries return flat output, persisting only attributes from the outermost level. Figure 7.3 presents the query that takes the four-levels of input (`RNCOP`) as well as the corresponding standard and shredded query plans.

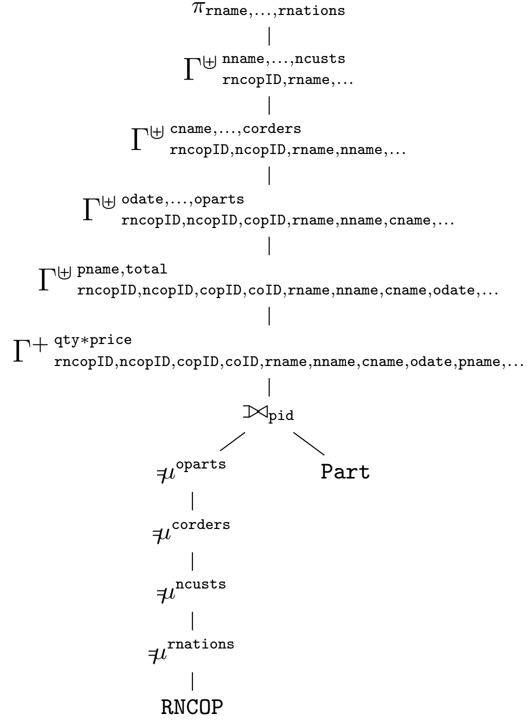
7.2 Experimental Setup

All nested TPC-H benchmark experiments were run on a Spark 2.4.2 cluster (Scala 2.12, Hadoop 2.7) with five workers, each with 20 cores and 320G memory. Each experiment was run as a Spark application with 25 executors, 4 cores and 64G memory per executor, 32G memory allocated to the driver, and 1000 partitions used for shuffling data. Broadcast operations are deferred to Spark, which broadcasts anything under 10MB. Total reported runtime starts after caching all inputs, with summary information on shuffling cost where relevant. All missing values marked with **F** correspond to a run that crashed due to memory saturation of a node.

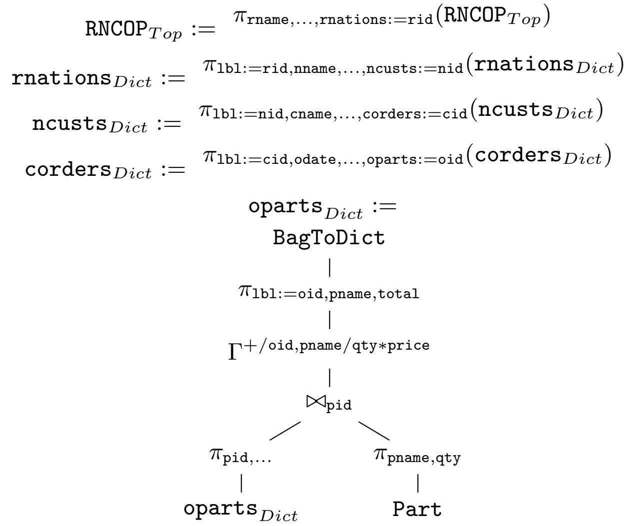
Datasets were generated using the skewed TPC-H data generator [66] with scale factor 100, producing a 100GB total. Zipfian distribution was used to produced skewed datasets; skew factor 4 is the greatest amount of skew, with a few heavy keys occurring at a high frequency. Non-skewed data is generated with skew factor 0, generating keys at a normal distribution as in the standard TPC-H generator.

```
for r in RNCOP union
 {⟨rname := r.rname, ..., rnations :=
  for n in r.rnations union
   {⟨nname := n.nname, ..., ncusts :=
    for c in n.ncusts union
     {⟨cname := c.cname, ..., corders :=
      for o in c.corders union
       {⟨odate := o.odate, ..., oparts :=
        sumBy^total_pname(
         for l in o.oparts union
          for p in Part union
           if l.pid == p.pid then
            {⟨pname := p.pname,
             total := l.qty * p.price⟩})⟩}⟩}⟩}⟩}
```

(a) NRC_{agg} program



(b) Standard compilation plan

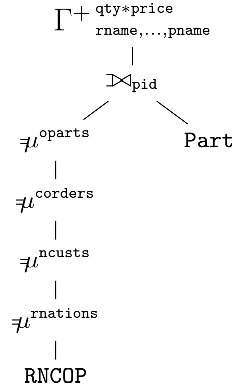


(c) Shredded compilation route plan

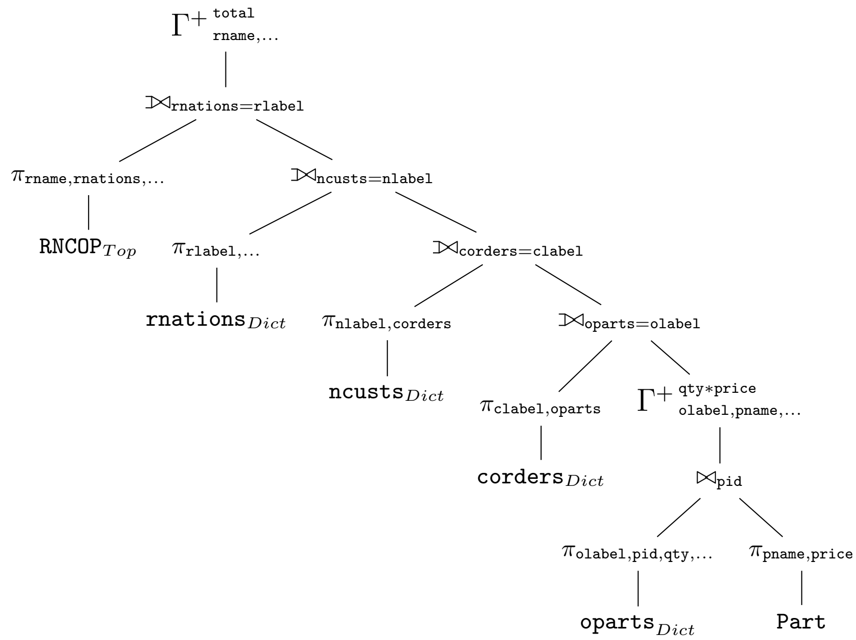
Figure 7.2: Nested-to-nested query with four levels of nesting.

```
sumBy^total_rname,...(
 for r in RNCOP union
  for n in n.rnations union
   for c in n.ncusts union
    for o in c.corders union
     for l in o.oparts union
      for p in Part union
       if l.pid == p.pid then
        {⟨rname := r.rname, ...
         total := l.qty * p.price⟩})
```

(a) NRC_{agg} program



(b) Standard compilation plan



(c) Shredded compilation route plan

Figure 7.3: Nested-to-flat query with four levels of nesting.

7.2.1 Systems Evaluated

The experiments compare the standard compilation route to the shredded compilation route, along with an array of external systems. The external systems were chosen based on their ability to perform at least one of the TPC-H benchmark queries, and include the SQL interface of SparkSQL [40], DIQL [52], Citus Postgres [51], and MongoDB [6]. Recall that TRANCE compiles to the SparkSQL Dataset API, so this provides a comparison between hand-optimized SQL queries in Spark and NRC_{agg} queries compiled into SparkSQL using our framework. DIQL is a recently-developed system that automates flattening procedures and also works on top of Spark. Citus is an object-relational approach using distributed Postgres instances in the background. MongoDB is the document store representative. All the queries for the external systems were manually written and optimized as much as possible. Relevant details of each of the experiments are outlined below.

SQL Queries in SparkSQL. SparkSQL provides an interface to execute distributed SQL queries on top of Spark (Section 2.6). Each query was written based on two restrictions. First, SparkSQL does not support explode (i.e., UNNEST) operations in the SELECT clause, requiring the operator to be kept with the source relation. This forces flattening for queries that take nested input. Second, joins and outer-joins cannot follow an explode statement, requiring the query to be written in the following form:

```
...
FROM (
  SELECT
  FROM Q
  LATERAL VIEW explode(Q.A) AS B
  -- cannot have here another join
) t1
LEFT OUTER JOIN Parts P ...
```

DIQL. DIQL is a query language and optimization framework that is deeply embedded in Scala. As mentioned in Section 2.4, this extends the decorrelation approach of [10] to work on top of distributed processing platforms. The syntax of DIQL fully supports all the queries in the TPC-H benchmark; however, this is an experimental

system and a bug in the backend meant only the flat-to-nested queries were supported. The DIQL Spark API requires Spark 2.4.3 and Scala 2.11.

Citus Postgres. Citus is a relational engine that uses a coordinator node to execute queries to distributed Postgres instance across worker nodes. Data import occurs by submitting the data to the coordinator node, which then distributes to each worker. Queries were written using both arrays and JSON with and without caching inputs. Caching and parallelization of inputs was increased based on the below configurations; however, better performance was achieved using defaults. The results are provided for the array-based queries without caching and with default worker configurations since these outperformed all other configurations.

```
shared_buffers = 80GB
effective_cache_size = 200GB
work_mem = 64MB
max_worker_processes = 20
max_parallel_workers_per_gather = 20
max_parallel_workers = 20
```

Distributed joins in Citus are trivial, collecting all data on the coordinator to perform the join before redistributing. Our experiments thus use hand-crafted optimization based on several caveats. First, Citus does not support nested subqueries in the target of a SELECT, failing with *could not run distributed query with subquery outside the FROM, WHERE and HAVING clauses.* Second, queries can be rewritten using GROUP BY and ARRAY_AGG, but joins between relations partitioned on different columns - known as complex joins in Citus terminology - are not supported; this fails with *complex joins are only supported when all distributed tables are co-located and joined on their distribution columns.* Outer joins can be done in a binary fashion with one table being a common table expression (CTE). For instance, the following query where t1 and t2 are partitioned on the join key but not on the join key for t3:

```
SELECT
  FROM (
    OUTER JOIN t1 and t2
  )
  OUTER JOIN t3
```

The result of the subquery (t1 join t2) will be collected entirely at the master and

then partitioned to workers according to the next join key. This is obviously inefficient and has restrictions logged in Citus as:

DETAIL: Citus restricts the size of intermediate results of complex subqueries and CTEs to avoid accidentally pulling large result sets into one place.

Third, left outer-joins between tables partitioned on different keys are not yet supported [1]. Finally, to avoid pulling data back to master and enable outer-joins between relations partitioned on different keys, execution plans were manually created where at each step an outer-join of two relations - partitioned on the same key - was performed. The results were then wrote back into a distributed materialized view partitioned on the next join key in sequence. This required the materialization of the entire flattened nested object to be repartitioned by `pid` before joining with `Part`. Each nested-to-flat query required 2 queries, and each nested-to-nested has one extra query for the final regrouping.

MongoDB. MongoDB is a document store that can use a central node to communicate with worker nodes that operate on distributed chunks of data, known as *shards*. Five workers were used as in the Spark and Citus setup. The queries were written following some restrictions. Only one collection can be sharded when performing joins and the inner relation must be local. The only join strategy iterates in parallel over the outer collection and performs lookups on the local, inner collection, which is the main bottleneck of this operation. We find that MongoDB has good performance for selective filters over a single collection, but is not designed for queries over multiple collections or even single-collection queries that return many documents. Nested collections formed using the `$push` accumulator are currently capped at 100MB; pipelines using more than 100MB will fail.

7.2.2 Additional Systems Explored

Several additional systems were explored that could not support the queries of the benchmark. These are outlined below.

[1] `https://github.com/citusdata/citus/issues/2321`

Rumble. Rumble, first mentioned in Section 2.8, transforms JSONiq to Spark and supports local and distributed execution. Problems were discovered running toy examples and doing data denormalization. Initially, outer-joins were not supported: `https://github.com/RumbleDB/rumble/issues/760`; we reported this and it was fixed, but now outer-joins with distributed collections are transformed into Cartesian products[2]. In general, Rumble does not provide several of the operations available in Spark, such as caching, schema handling.

Zorba. Zorba is a query compiler for both XQuery and JSONiq. It does not support distributed data and has also not been maintained in the past four years [67].

MonetDB. MonetDB [68] is a high-performance database engine based on column storage. The system, however, has no support for collection types and operations. The system does support JSON operations over strings, but there is no easy way to transform tables to JSON objects; manually creating JSON strings throws errors.

Cockroach. CockroachDB is a distributed SQL database. It does not support nested collections [69].

VoltDB. VoltDB [70] is an in-memory database that provides pre-compiled transactions for Java. JSON is supported but the application must perform the conversion from an in-memory structure to the textual representation. In addition, there is a size limit for JSON values; the VARCHAR columns used to store JSON values are limited to one megabyte (1048576 bytes). In summary, JSON support allows for augmentation of the existing relational model with VoltDB; however, it is not intended or appropriate as a replacement for pure blob-oriented document stores.

YugabyteDB. YugabyteDB [71] is a distributed SQL database that includes support for collections. The performance was too poor to explore further; for example, the following query took four minutes with only 18760 orders tuples and 2500 user tuples:

```
SELECT users.id,
  (SELECT ARRAY_AGG(orders.id)
   FROM orders
   WHERE orders.user_id=users.id)
FROM users
```

[2] Accessed 10 September 2020.

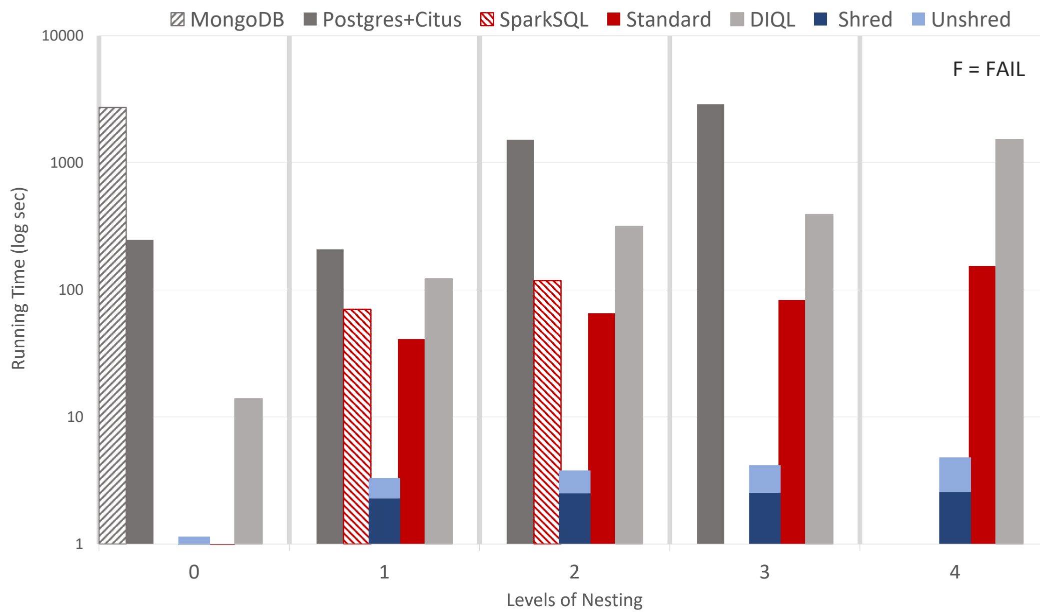
7.2.3 Experimental strategies

The experiments refer to the various compilation methods in TraNCE. Standard denotes the standard compilation route and produces results that match the type of the input query. Standard also serves as a means to explore the general functionality of flattening methods within the same system as shredding techniques. Shred denotes the shredded compilation route with domain elimination, leaving its output in shredded form. Shred^{+U} represents the time required to unshred the materialized dictionaries, returning results that match the type of the input query. Shred^{+U} is often considered in combination with Shred to denote total runtime of the shredded compilation route when the output type is nested. Since SparkSQL outperformed all the other external systems, it is used as the representative competitor for some of the experiments.

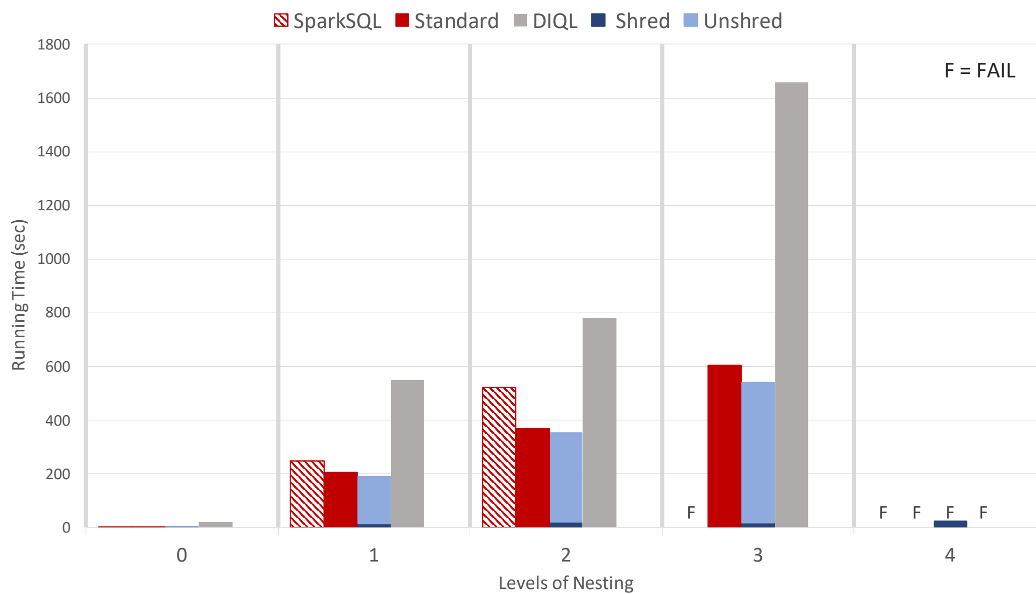
The above strategies do not implement skew-handling; $\text{Standard}_{\text{skew}}$, $\text{Shred}_{\text{skew}}$, and $\text{Shred}^{+U}_{\text{skew}}$ are used to denote the skew-handling variations for the TraNCE runs. The performance of each query is measured based on the optimal plan for a given strategy, which at minimum includes pushing projections and domain-elimination (Section 4.3). Any further optimizations are described within the context of each experiment.

7.3 Experimental Results for Non-Skewed Inputs

The flat-to-nested queries are used to explore building nested structures from flat inputs when the data is not skewed. Figure 7.4a displays the results for MongoDB, Postgres Citus, DIQL, SparkSQL, Standard, Shred, and Shred^{+U} for the narrow flat-to-nested queries of the TPC-H benchmark. Figure 7.5 includes the results for all systems for nested-to-nested and nested-to-flat queries, which evaluates the impact of processing nested inputs. Given the performance for all systems is worse for wide tuples, the flat-to-nested wide variants were not explored for MongoDB and Citus. Figure 7.4b displays the results for DIQL, SparkSQL, Standard, Shred, and Shred^{+U} for the wide flat-to-nested queries. The wide variants for the nested-to-nested and nested-to-flat queries were also not explored due to poor performance. Since the MongoDB and Postgres Citus queries could only be preprocessed with projections pushed,

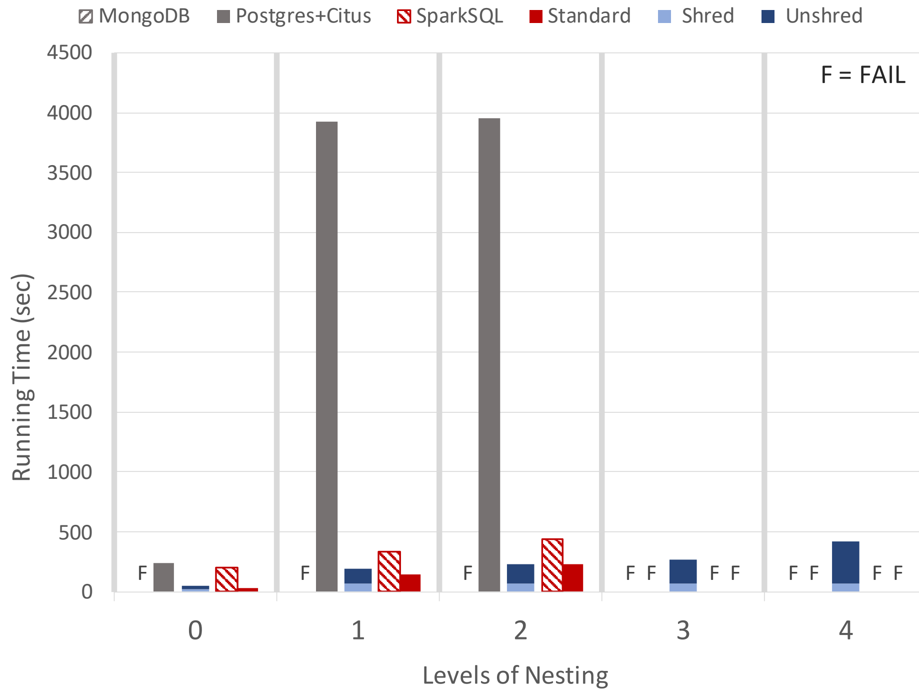


(a) Narrow schema - log scale

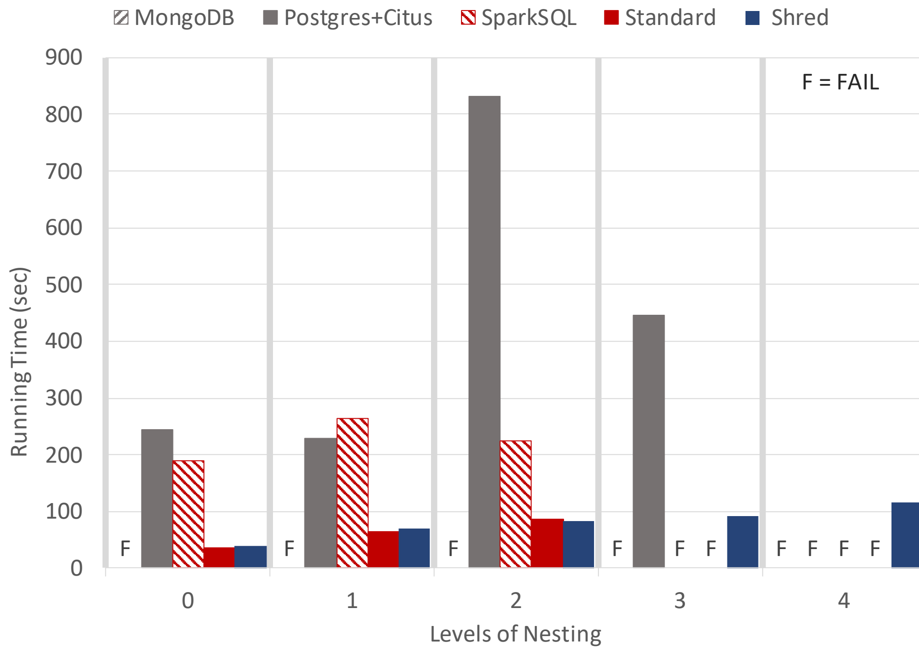


(b) Wide schema

Figure 7.4: Performance comparison of flat-to-nested queries including all competitors. MongoDB and Citus failed for all wide variants so are not included.

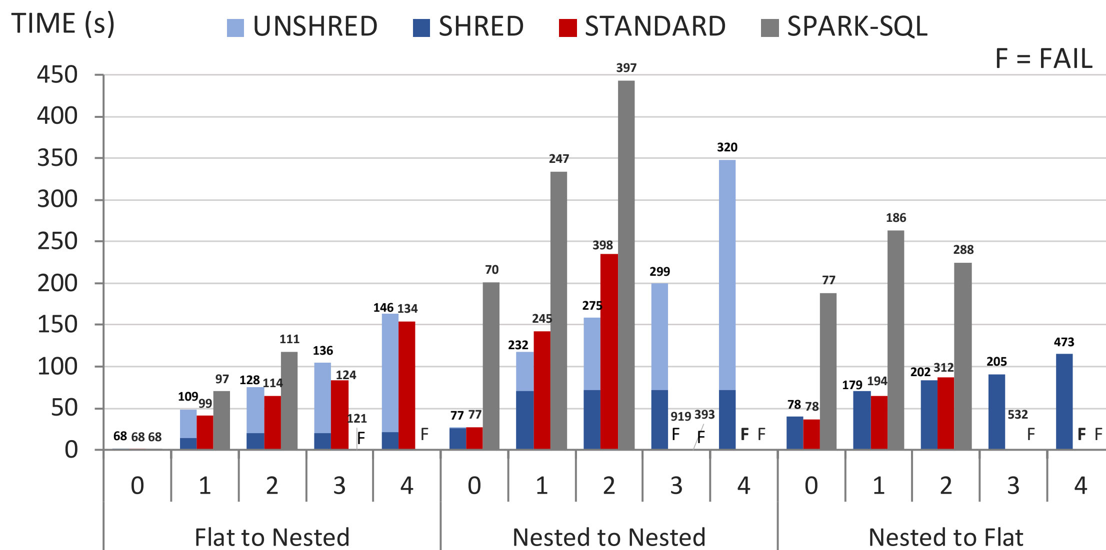


(a) Narrow nested-to-nested

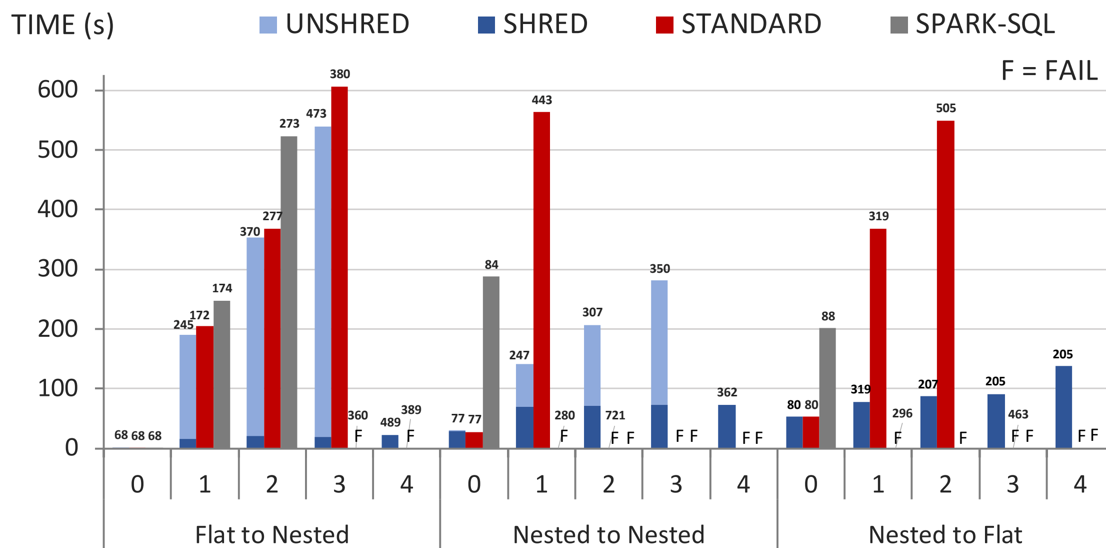


(b) Narrow nested-to-flat

Figure 7.5: Performance comparison of narrow nested-to-nested and nested-to-flat queries for all competitors.



(a) Narrow schema



(b) Wide schema

Figure 7.6: Performance comparison of the narrow and wide benchmarked TPC-H queries with varying levels of nesting (0-4). Each run is marked with total shuffle memory (GB).

these systems take the materialized narrow flat-to-nested query as input. The other systems take the materialized wide flat-to-nested query as input to better explore the effects of projection.

All query categories and variants were explored for the best performing external system, SparkSQL, as well as for STANDARD, SHRED, and SHRED^{+U}; these results for all levels of nesting are presented in Figure 7.6, which includes the total shuffled memory (GB) for each run. If a job crashes at a particular nesting level, any further total shuffle memory is not reported.

The experiments overall present advantages for using the shredded representation in both the succinct representation and optimization opportunities. The results for each query category are discussed below.

7.3.1 Flat-to-nested

The flat-to-nested queries are used to investigate the impact of building nested collections from flat relations in distributed frameworks. The discussion begins with the base case: performing an operation on a flat relation with no nesting in the output, followed by a discussion on the performance of both nested output and the shredded representation.

No Nesting Base Case. For level 0 with no nesting (Figure 7.4), the wide variant is a sanity check that measures the time to return the `Lineitem` relation (75G); all systems have comparable runtimes. The narrow variants measure the time to project the `pid` and `qty` attributes from `Lineitem`. The systems diverge at level 1 where MongoDB takes 45 minutes. Citus has the second longest runtime, but is still 11× faster than MongoDB. All the other systems have comparable runtimes for both variants of level 0; since all run on top of Spark, the projections are performed locally at each partition and any additional compute time for the projection is amortized by reducing the amount of the data returned.

Nested Output. All systems that return nested output have increasingly worse performance as nesting deepens (Figure 7.4). MongoDB fails to run for any nesting in the output. The runtime of Citus shows no increase for one level nesting, but exhibits 11×

for three levels and then crashes at four levels. An interesting result is that Citus is able to survive to three levels of nesting when SparkSQL can only produce two levels. SparkSQL is unable to perform past two levels of nesting for both narrow and wide variants, which is attributed to the inability of the optimizer to merge join and nest operations into cogroup (Section 3.4). Though SparkSQL exhibits slower runtimes, the total shuffled memory is lower. This could mean that the Catalyst optimizer is performing shuffle-reduction heuristics that do not result in better overall runtime.

The use of cogroup proves important for scaling to deeper nesting, smaller numbers of top-level tuples, and moderately sized inner collections - as seen in the survival of Standard, Shred^{+U}, and DIQL for narrow variants. Note that Standard and DIQL have similar execution strategies, with the increase in runtime of DIQL attributed to the use of RDDs which have a higher memory footprint than Datasets (Section A.1). While Standard performs slightly better than Shred^{+U} for the narrow variant, Shred^{+U} shows better scaling to wide variants with a 1.5 minute advantage over Standard. All systems that return nested output fail the stress-test for deep nesting with wide variants. This affirms that large-sized inner collections and top-level distribution strategies are failing points. Shred is the only remaining survivor of the deeply nested wide variant.

Shredded Representation. Shred runs to completion for all levels, remaining constant after the first level and exhibiting nearly identical runtimes and max data shuffle for both narrow and wide variants (Figure 7.6). Shred shows 6x improvement for narrow queries and 26x for wide queries in comparison to the best performing external system, SparkSQL. Shred^{+U} and Standard have comparable runtimes overall, consistent with their similar execution strategies (Section 7.1.1). These methods also shuffle 20x more than Shred.

The only method to use the shredded representation throughout is able to survive the stress-test. It is true, however, that the shredded representation may not be practical for immediate use in existing applications. Other than TraNCE, there is no alternative system or application that can operate on this representation directly. This means that returning the nested output is currently a requirement in many cases, such as returning a JSON-formatted data source to a web frontend. These results show that in such a case there is no overhead in using the shredded representation. This is an important finding since the flat-to-nested case can be a worse-case scenario for shredding:

the query is executed only to group everything up in the unshredding phase. These results show that the joins in unshredding do not add anymore cost than existing techniques; in fact, the results show that the shuffling costs cannot be avoided for any processing technique that returns nested output.

7.3.2 Nested-to-nested

The nested-to-nested queries explore the effects of aggregations on non-skewed data. The materialized result of the flat-to-nested wide variant is used as input in order to evaluate the impacts of projections. MongoDB and Citus use the narrow variants since they are unable to produce the wide variant, as noted above. Due to the bug noted in Section 7.2.1, DIQL is not evaluated for queries that take nested input.

No Nesting Base Case. Level 0 with no nesting (Figure 7.5a) joins `Lineitem` with `Part` and performs a sum aggregate and grouping by `pname`. This upper-bounds the result to the size of `Part`. MongoDB is unable to perform at all, due to the large data size and poor join performance. Citus with pre-projected input exhibits similar performance to SparkSQL for the no nesting case. SparkSQL is over 7× worse than the other systems that use Spark (Standard and Shred) for the narrow variant and over 10× as worse for the wide variant, suggesting missed projection opportunities (Figure 7.6). Standard and Shred^{+U} have identical execution strategies and runtimes for 0 levels of nesting. These methods report the best overall runtime for 0 levels, with Shred^{+U} eventually outperforming Standard for deeper levels.

Nested Output. All systems that return nested output have increasingly worse performance for deeper levels of nesting (Figure 7.5a). Citus runtime increases up to 16x for shallow nesting and fails or deeper levels. The first two levels return the same runtime in Citus, suggesting a bottleneck in the lower-level join and aggregation. The pre-projected input for Citus has kept flattening relatively light-weight for shallow levels; however, the small number of top-level tuples for deeper nesting crashes the system. While Citus and SparkSQL had similar results for no nesting, SparkSQL has a 9× advantage for shallow nesting. In comparison to other Spark-based systems, SparkSQL has consistently worse runtimes though the same total amount of shuffled memory. For wide tuples, SparkSQL does not survive with any nesting (Figure 7.6) and Standard only survives to one level. The wide variant pressures the system when

flattening techniques are employed; for example, at two levels the flattened result will pass the join of `Customer`, `Order`, `Lineitem`, and `Part` to the aggregation using all the upper-level attributes in the key, which is a very expensive operation.

In the narrow case, Standard survives the aggregation but fails for smaller numbers of top-level tuples. Shred^{+U} shows a linear drop in performance as the number of top-level tuples decreases from 150 million to 25 tuples, though at a lower rate than the flattening methods. Shred^{+U} survives all levels of nesting for the narrow variant and up to three levels of nesting for the wide variant. For 5 top-level tuples, Shred^{+U} fails to maintain large inner collections with this poor distribution strategy and crashes due to memory saturation. Shred^{+U} provides up to 25x performance advantage over Citus and 7.5x that of SparkSQL, and survives to deeper nesting when other fail.

Systems that use flattening methods (SparkSQL and Standard) consistently fail when operating on nested inputs, where the flat input case is able to survive. The nested-to-nested queries fail at the most memory-intensive stage of the flat-to-nested queries. Where the flat-to-nested query is able to iteratively group individual relations, the flattening technique carries around additional data while re-grouping that leads to a tipping point for the memory-intensive stages. The survival of Shred^{+U} further supports this observation, where the shredded outputs are iteratively re-grouped resulting in 3× less shuffled memory.

Shredded Representation. Shred runs to completion for all levels showing similar runtimes and data shuffle for both narrow and wide variants. The shredded representation has a 55× advantage over Citus and survives when all others methods fail. The Shred runtime is constant for nesting levels, which is about 2.5× that of level 0. The additional time is the repartitioning of the lower level dictionary by label. Constant runtime is achieved by localizing the operation to the lowest level dictionary, such that adding on upper-level shredded representations adds no additional time. Localized aggregation also reduces unshredding time. In comparison to previous results without aggregation, the localized operation has reduced the unshredding time for the wide variant. These are two important advantages of the shredded representation. Operations can be applied locally to a given level while maintaining a light-weight association to the parent level. When these operations lead to a reduction of data at any given level, the workload to return nested outputs is lessened.

7.3.3 Nested-to-flat

The nested-to-flat experiments evaluate operations that occur over multiple levels of the input to return flat output. The materialized result of the flat-to-nested wide variant is used as input as in the previous experiment, following the same conditional evaluation as the prior experiment: MongoDB and Citus use the narrow variants and DIQL was not evaluated due to a bug. As in the prior experiment, MongoDB does not finish for any of the queries. Since all output types are flat for this experiment, there are no results for $\text{Shred}^{+\text{U}}$. The query and performance for level 0 are the same as the nested-to-nested results (Figure 7.5b).

Nested-to-flat queries persist only the top-level and lower-level attributes thus introducing more projection opportunities. This is an advantage for flattening methods that introduces data reduction opportunities without the need for any additional joins. Alternatively, the shredded representation must reassociate levels to return flat output and this increases the number of joins in the query plan.

Nested Input. When operating on nested inputs, all systems tend to have better performance for flat outputs. Citus performs slightly better than SparkSQL for one level of nested input. The runtime for Citus decreases slightly from 0 to 1 levels, then increases 3.5× for 2 levels, decreases again for 3 levels, and then fails for 4 levels. This behavior suggests that runtime is proportional to the size of the aggregated result. At one level, more reduction occurs since the key is on `odate`. There is less reduction for two levels when aggregating by `cname`, but then more reduction for three levels when aggregating by the 25 `nname` values. The failure at 4 levels suggest there is a point at which poor distribution and flattening are detrimental even when the output is small.

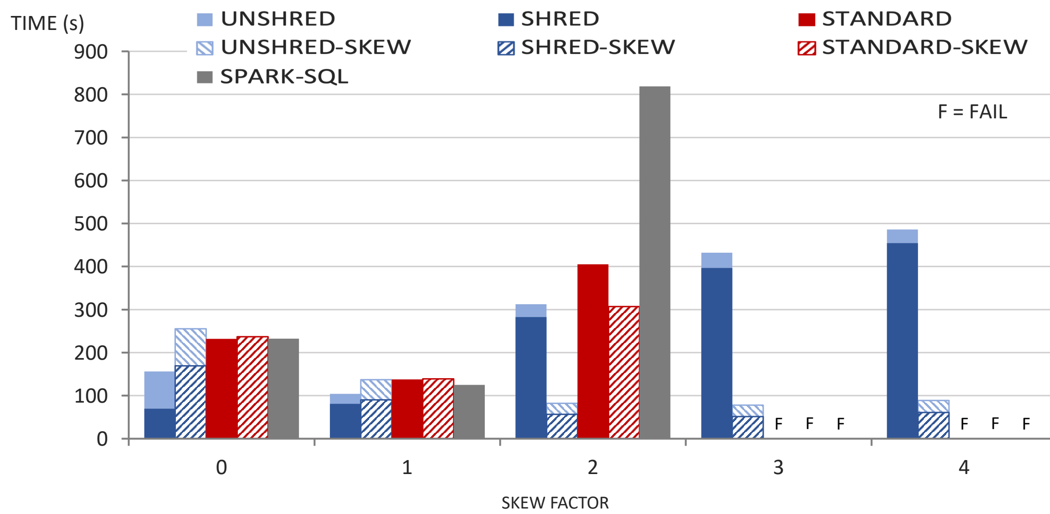
SparkSQL runtimes are consistent up to two levels of nesting and then fail for deeper levels when the number of top-level tuples limits distribution. For wide tuples, SparkSQL cannot complete even for one level of nesting. SparkSQL was actually unable to survive past two levels of nesting for any of the query categories, suggesting that the performance is dependent on a reasonable distribution strategy. The results also are consistent with missed projection opportunities that were observed in the prior two experiments. Standard is unable to run for the small numbers of top-level tuples for both the narrow and wide case, implying that this is a general issue for flattening methods on Spark.

In addition to projection opportunities, aggregations could not be pushed past the nested join. Since the top-level information must be included in the aggregation key, aggregation pushing is not possible for this case. Another potential optimization is local aggregation (Section 5.2). Due to the nature of the TPC-H data, the flattening methods do not benefit from local aggregation since both intermediate aggregations would work on key attributes. However, local aggregations are the key to the success of Shred, which outperforms all other methods.

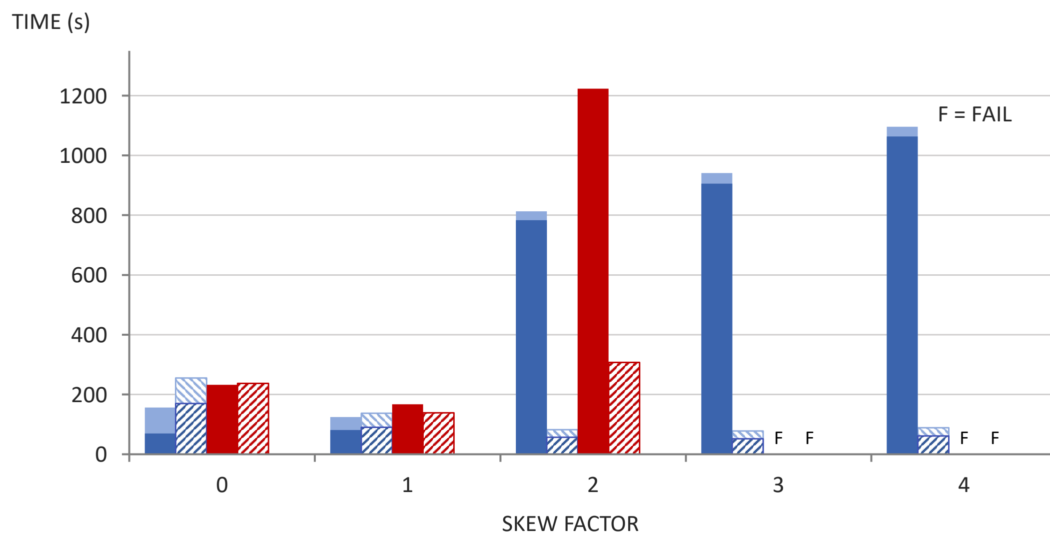
Shredded Representation. Though the nested-to-flat queries are thought to be a disadvantage for shreddding methods, the results here highlight how the shredded representation provides optimization opportunities that would otherwise not be accessible to alternative methods. Shred outperforms Standard with a 6x runtime advantage, over a 2x total shuffle advantage for wide queries, and runs to completion even when other systems fail. At two levels of nesting, the execution of Shred begins with a localized join between the lowest-level Lineitem dictionary and `Part`, then aggregates the result. Unlike the flattening procedures, these operations avoid carrying redundant information from top-level tuples, thereby reducing the lowest-level dictionary as much as possible. The evaluation continues by dropping all non-label attributes from the first-level Order dictionary and then joining this with the aggregated result. Both the lowest-level and first-level dictionary are reduced as much as possible, which results in a cheap join operation. This benefit persists even as the number of intermediate lookups increase for deeper levels of nesting, exhibiting resilience to the number of top-level tuples. By the time the final result is joined with the final top-level, the light-weight intermediate is aggregated a final time to produce an output with only 5 tuples. The shredded representation introduced optimization opportunities that avoid the expense of the additional joins.

7.4 Experimental Results for Skewed Inputs

The narrow variant of the nested-to-nested TPC-H query with two levels of nesting is used to evaluate the skew-handling procedure. The materialized flat-to-nested narrow query with two levels of nesting (`COP`) is used as input. All experiments use the sampling-based heavy key identification method, additional identification methods are explored in later chapters with biomedical queries. The TPC-H skew ex-



(a) Skew-unaware with aggregations pushed, skew-aware without aggregations.



(b) Without aggregation pushing

Figure 7.7: Performance comparison with and without skew-resilience for increasing amounts of skew.

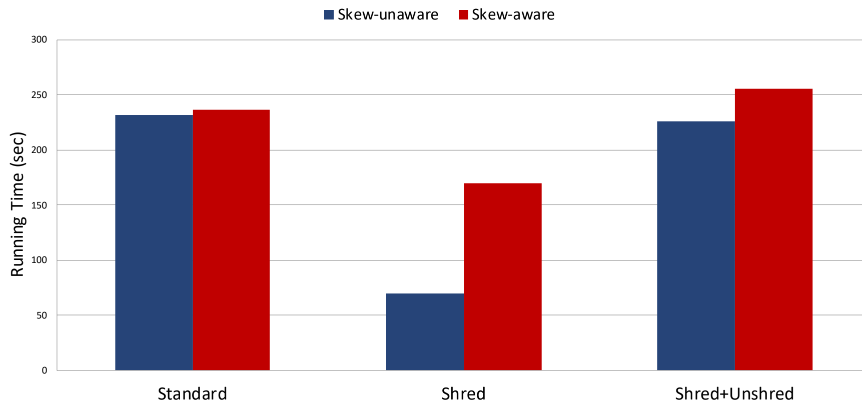


Figure 7.8: Heavy key identification overhead for the sampling method.

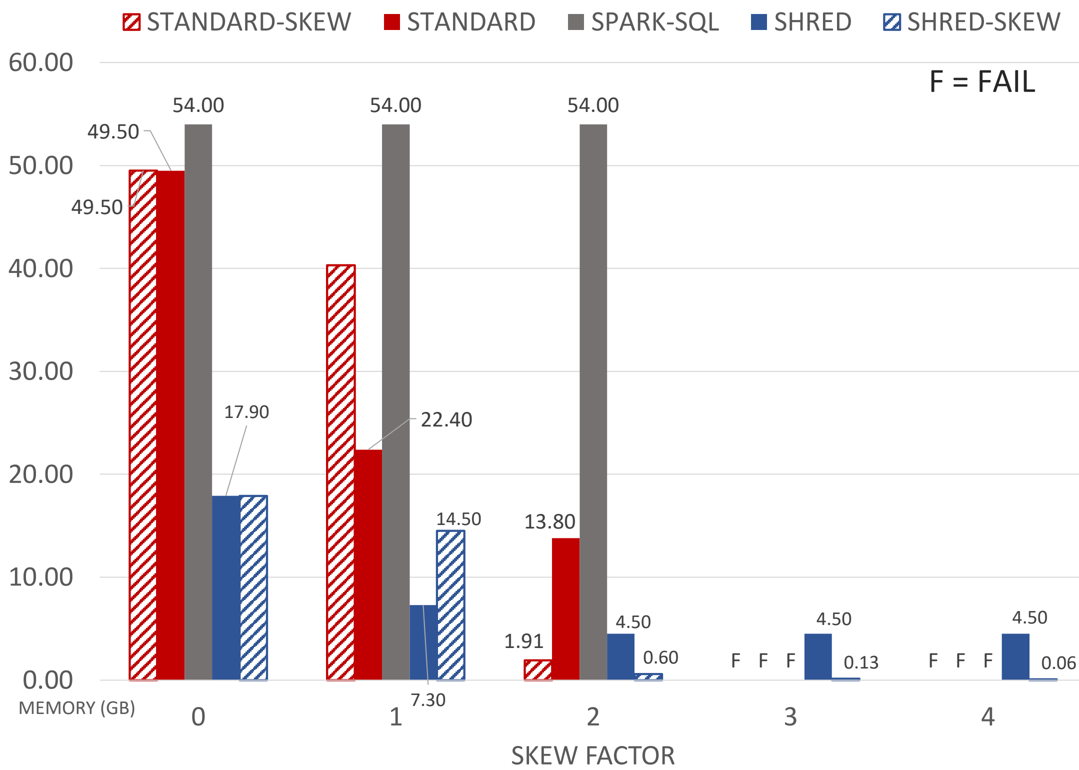


Figure 7.9: Amount of shuffle related to the join on skewed data.

periments compare SparkSQL and the TraNCE compilation routes for increasing amounts of skew in nested data processing, including Standard, Shred, $\text{Shred}^{+\text{U}}$, $\text{Standard}_{\text{skew}}$, $\text{Shred}_{\text{skew}}$, and $\text{Shred}^{+\text{U}}_{\text{skew}}$. The runtime results are presented in Figure 7.7a.

7.4.1 Skew-handling Overhead

For skew factor 0, the difference between the skew-aware (Standard, Shred) and skew-unaware ($\text{Standard}_{\text{skew}}$, $\text{Shred}_{\text{skew}}$) methods is the time to perform the sampling-based heavy key identification. There are two skew operations that will trigger heavy key identification. The first is a join that is specific to all compilation routes, and the second is the repartitioning from `BagToDict` that is specific to the shredded compilation route. The first identification takes $\approx$ 10 seconds, which is observed in the difference between Standard and $\text{Standard}_{\text{skew}}$. The second identification adds 1.5 minute overhead to the $\text{Shred}_{\text{skew}}$ in comparison to Shred. This calculation takes a long time due to the size of the lower-level dictionary. Since the calculation is expensive when data is large, the slice technique was also evaluated. The slice method reduced the heavy key calculation by 20 seconds. However, this technique was not accurately estimating the heavy keys in the presence of skew and did not improve performance over sampling techniques in the presence of skew. In summary, the heavy key identification is an investment that will be paid off even for low amounts of skew, as described in the next section.

7.4.2 Increasing Amounts of Skew

The results show that even for low amounts of skew the skew-aware methods of the standard and shredded pipeline outperform the skew-unaware methods, thus amortizing the heavy key calculation overhead. For a moderate amount of skew (skew-factor 2), $\text{Standard}_{\text{skew}}$ has a 5x performance gain over Standard. At greater amounts of skew, neither Standard nor $\text{Standard}_{\text{skew}}$ run to completion, crashing due to memory saturation. $\text{Shred}_{\text{skew}}$ outperforms $\text{Standard}_{\text{skew}}$ by 4x and Standard by 22x, whereas $\text{Shred}^{+\text{U}}_{\text{skew}}$ shows 3x and 4x improvements, respectively. These results are based on Figure 7.7a, which used the optimal plan for each method.

For the skew-unaware methods the optimal plan includes local aggregation, whereas the skew-aware plan does not include local aggregation. Figure 7.7b presents the results of all TraNCE compilation routes without local aggregation, where skew-unaware methods have a 3× drop in performance. This behavior is a product of the TPC-H data generator, which produces skew by duplicating values. Local aggregation reduces the amount of duplication, which diminishes the signal of heavy keys without fully handling data skew. There is an interesting trade-off between data reduction optimizations and the identification of heavy keys. While local aggregation is beneficial for non-skewed data and skew-unaware methods, the ability to identify heavy keys and properly distribute their values is more important for skew-handling methods.

The handling of heavy keys and broadcasting their values greatly reduces the overall memory of an application. Figure 7.9 displays the associated shuffled data for the skew operation for all methods. This excludes unshredding since the skew operation is performed in Shred and $\text{Shred}_{\text{skew}}$. There are no `COP` shuffle results for the standard compilation route since query fails during execution while attempting to flatten `COP`. SparkSQL survives flattening for high levels of skew, but fails while performing the join with `Part`. The shuffling of the standard compilation route shows that at lower levels of skew the local aggregation is beneficial. At higher levels of skew the local aggregation reduces the data in the skew-handling variation to 14G in the standard compilation 4.5G for the shredded compilation. The skew-aware shredded compilation is able to reduce this to only megabytes for high amounts of skew, leading to 74× less shuffle than the skew-unaware compilation route. The performance gain of the skew-aware shredded pipeline over the skew-aware standard pipeline is attributed to the shredded representations ability to better support skew-aware operations.

7.4.3 Skew-handling for the Shredded Representation

The skew-aware shredded pipeline outperforms all strategies. $\text{Shred}_{\text{skew}}$ shows a 14x improvement over Shred, and $\text{Shred}^{+\text{U}}_{\text{skew}}$ shows a 10x improvement over $\text{Shred}^{+\text{U}}$. Both versions of the shredded compilation route are able to run to completion for all skew factors, highlighting the natural skew-handling abilities in the shredded representation. As in the non-skewed experiments that operate on nested data, the shredded representation localizes aggregation to the lower-level dictionary thereby decreasing

skew. This is seen by the decrease in Shred^{+U} and $\text{Shred}^{+U}_{\text{skew}}$ for higher amounts of skew, which benefit from reducing the size and skew involved in regrouping. The skew-aware pipeline benefits further from localizing an aggregation to a specific shredded representation since it reduces the time to calculate heavy keys in `BagToDict`, which was previously causing a spike for the non-skewed data (Section 7.4.1).

Interestingly, $\text{Shred}_{\text{skew}}$ and $\text{Shred}^{+U}_{\text{skew}}$ show improved performance as skew increases. This suggests that as skew increases, the shredded pipeline is not affected by the size of the bags associated to the heavy keys, with skew-resilience ensuring these bags will remain distributed. These results also validate the assumption that the set of heavy keys will be small, supporting the use of broadcast-based methods for the skew-aware join. Overall, the skew-aware, shredded pipeline outperforms all other methods, gracefully handling increasing skew, even when the flattening methods are unable to run at all.

7.5 Experiments Summary

This chapter uses a novel nested benchmark based on TPC-H to compare performance of large-scale processing of nested data in existing systems. The use of the standard and shredded compilation routes provides an evaluation of the shredded representation compared to flattening techniques implemented in the same system. The results confirm there are outstanding performance issues for distributed, nested data processing in current systems and show how shredding could be used to increase performance.

Flat-to-nested, non-skewed data. The flat-to-nested experiments on non-skewed data sets the stage for evaluating the performance of systems that process nested data, showing clear performance issues with current systems even for flat input datasets. Though MongoDB is a popular choice for nested datasets, the system is not well-equipped for large-scale data especially when nested joins are involved. Nested joins in general appear to be a fault point for many of the systems, especially under "stressful" conditions. If all systems fail under such conditions, then there is much to gain from the shredded representation. Shredding can provide low execution costs and leave room for downstream computation, which can further reduce data size and reassociation costs.

Nested-to-nested, non-skewed data. The nested-to-nested queries on non-skewed data highlight many advantages to shredding over flattening methods. There are major performance issues for current systems for both processing and producing nested collections. The shredded representation is light-weight and can localize operations to reduce otherwise memory-intensive tasks. These shredding techniques can reduce the overheads that come from returning nested outputs. However, these experiments measure runtime assuming the data has been pre-shredded. While the shredded representation is produced from an upstream process in these queries, there will be cases where the dataset needs to be shredded before the query can be performed. Further exploration into value shredding techniques and the overhead in shredding pipelines is necessary. With that in mind, the results here suggest that the shredded representation could be the key to scaling to deeper levels of nesting. If shredding techniques produce more scalable results, than it would be advantageous for applications to provide and operate on this more succinct representation.

Nested-to-flat, non-skewed data. The nested-to-flat results with non-skewed data show that existing systems are more equipped to deal with flat outputs than nested outputs. The flattening methods prove to be sensitive to small numbers of top-level tuples that result in poor distribution. While flattening methods can project away intermediate values, there are many missed opportunities for local aggregation and pushing of aggregations. The shredded representation can introduce such opportunities for optimization in comparison to flattening methods. The shredded representation opens up opportunities for applying optimizations, previously specific to relational settings, to nested data.

There is more to be investigated regarding queries that return flat output. For instance, a program that operates on flat inputs could build up many nested intermediates, only to later tear them down and return flat output. Normalization techniques would inline such programs to remove any intermediate nesting, while the shredded representation would use the sequential materialization strategy. The sequential strategy limits optimization opportunities that could otherwise occur between query plans if one, normalized query was considered. The results here show that the shredded representation presents additional optimizations that normalization may otherwise throw away - these trade-offs should be investigated.

Skewed Inputs. The skew-handling abilities are evaluated for TraNCE and Spark-

SQL, as the representative external system. The results show that heavy key identification in TraNCE can add compute time, but this pays off when the data is skewed. The skew-handling methods perform better in the presence of skew. The shredded representation brings more advantages to handling skew, scaling to deeper levels of nesting. The duplication in the TPC-H data generator presents an interesting detail about the interaction of local aggregation and heavy key identification; the skew-aware methods do not benefit from local aggregation because it disrupts the identification of heavy keys. Local aggregation could be of use to heavy key calculation, but it would need to occur after the heavy key identification of the skew operator. Further exploration is required to understand how these methods can be mutually beneficial.

Additional investigation for outer-joins in skew-handling and the caching of heavy keys should be considered. PRPD techniques have poor performance for outer-joins [72]; however, this has not been evaluated for nested collections. Outer-joins are essential for producing correct answers when operating on and returning nested collections; however, the current experiments do not capture the issues with scale. The flattening of the skewed inner collections in the standard compilation route crashes the application before the skew-aware outer-join can be executed and the shredded compilation route does not require outer-joins. While further investigation is needed to assess the scaling issues with the outer-join in the standard compilation route, the focus on nested collections suggests that there could be larger burdens to consider - and these are burdens that the shredded representation is able to avoid.

Shredding Overall. While the results show benefits to the shredded representation, there is more to consider for the practical use of shreddding in current systems. Many existing applications operate on the nested representation and though the experiments show no overhead to returning nested output in the shredded route, further investigation is required to understand how current systems can support the shredded representation. In the case of shredded inputs, the overhead associated with value shredding, how this process can be optimized, and how applications can return shredded representations to support downstream processing of shredded queries should be explored. It is important to understand how existing systems could support shredded representations internally and the design challenges involved for such extensions. For example, partial shredding can be used together with flattening techniques to allow flexibility in the representation based on collection size or a related cost-model.

Chapter 8

Biomedical Analysis and Performance in TraNCE

The previous chapter evaluated the performance of TraNCE and other nested data processing systems on a synthetic, micro-benchmark. This chapter focuses on the performance of these systems on real, biomedical datasets and applications. The biomedical use cases assess the expressiveness of the source language and the practical use of the framework for real-world workflows. Attention is paid to the use of the shredded representation for modern data analysis pipelines. The chapter begins with the relevance of biomedical data analysis for large-scale nested data processing and presents a running example. An overview of the biomedical data sources is then provided (Section 8.2). Three applications are then presented. The first application (Section 8.3) focuses on restructuring and exploration, such as requests a clinician would make from a user interface. The second application (Section 8.4) implements a multi-omics analysis pipeline, which is based on a research study that identifies driver genes in cancer [73]. Section 8.5 presents a single-omics analysis that builds mutational *burden-based* feature sets that are used for downstream classification. The chapter concludes with some additional experiments that highlight framework scalability and skew-handling techniques when applied to biomedical datasets.

The results of the biomedical use cases can be summarized as follows:

- Clinical exploration programs fall into the nested-to-nested queries with wide tu-

ples category, applying additional operations that lead to data reduction. Similar to the corresponding results with TPC-H data, the results show that flattening methods scale poorly and the shredding technique can return nested data without compromising performance.

- The programs of the driver gene analysis present a pipeline that operates on nested collections to eventually return flat output. These results show that the shredded compilation is beneficial even for smaller datasets and is the key to scaling for larger datasets.
- The burden-based programs provide an example of how TraNCE can be used to construct feature vectors that are used in classification tasks, generating notebooks that can interface with external Python libraries.
- Additional performance results are explored showing benefits to the shredded compilation route for scaling with respect to data size and cluster size. The heavy key identification methods are also explored for the biomedical datasets, highlighting benefits to the slice procedure.

8.1 Background

Biological data comes in a variety of domain-specific formats [74]. In addition, the affordability of genomic sequencing, the advancement of image processing, and the improvement of medical data management have made the biomedical field an interesting application domain for integrative analyses of complex datasets. Targeted medicine is a response to these advances, aiming to tailor a medical treatment to an individual based on their genetic, lifestyle, and environmental risk factors [75]. Analyses that combine molecular measurements from multi-omics data provide a more thorough look at the disease at hand, and the relative impacts on the underlying system. The reliability of targeted treatments is dependent on multi-modal, cohort-based analyses.

Targeted medicine has improved data management and data collection within medical institutions, which are now capable of producing biomedical datasets at outstanding rates. For example, the sequence archive from the NIH has exhibited exponential growth in less than a decade [76]. In addition, these efforts have also spurred consortium dataset collection and biobanking efforts [77]. These are consolidated data sources from hundreds-of-thousands of patients and counting, such as 1000 Genomes

[78], International Cancer Genome Consortium (ICGC) [79], The Cancer Genome Atlas (TCGA) [80], and UK BioBank [81]. This scenario has introduced a demand for data processing solutions that can handle such large-scale datasets; thus, scalable data integration and aggregation solutions capable of supporting joint inference play a key role in advancing biomedical analysis.

Biomedical data analysis workflows can range from a set of complicated scripts to standardized pipelines based on workflow languages [82]. A wide range of tools are available to assist biological analyses. Workflow languages can be executed on a workflow engine that eases the process of connecting many external software systems while producing repeatable analyses. Some workflow engines are geared toward a certain kind of biological analysis, such as Galaxy [83], Cromwell [84], and Arvados [85]; other workflow engines are less specific to the type of analysis, such as Taverna [86]. Workflow languages describe imperative pipelines that require manual optimizations to each individual pipeline component. In contrast, high-level, declarative languages better insulate pipeline writers from platform details, while also providing the ability to leverage database-style query compilation and query optimization techniques.

Biological analysis platforms can also come in the shape of query languages, such as GenoMetric [87] and analysis-specific APIs, such as Hail [88], Adam [89, 90], and Glow [91]. These provide advantages for a particular class of transformations, but would not suffice for pipelines that integrate a variety of relational and nested data types.

Data analysis in biology - with or without the use of a workflow engine - is a laborious process that requires the connection of several data types, software components, and technologies. There are several state of the art approaches to modern biological data analysis. Some solutions provide an easier way to interact with a certain data type and analysis, others are more agnostic to the type of data and the analysis type. Regardless of scope, current approaches attempt to address issues related to diverse datatypes, scalability, and external software.

When dealing with diverse data formats some more recent platforms often work within the nested relational data model [7, 87, 88, 92]. These languages follow a dialect of SQL, which can make comprehension style queries cumbersome to express leading to performance issues related to flattening and programming mismatches mentioned in the introduction.

The desire to scale has made distributed platforms common targets for biological analysis. Non-expert users have difficulty interacting with distributed computing frameworks, so solutions come as high level languages. GenoMetric works on top of Apache Pig, which provides a high level language - Pig Latin - for defining MapReduce jobs. Apache Pig also provides a set of rules for enabling user-desired optimizations. Hail is a Spark framework for exploring and analyzing genomic data. ADAM provides an Apache Spark toolkit for running distributed genomic analysis. Beyond distribution, local parallelization can be manually specified in the workflow languages of Cromwell, Galaxy, and Arvados.

The interaction of external software is often approached by the wrapping external tools [93, 86, 85]. The GenoMetric [87] query language requires materializing the output to a standardized format for consumption by downstream processes.

There is much to explore in the area of using declarative query techniques in biomedical pipelines, especially for the integration of complex data types. This thesis explores several use cases related to biomedical analysis, paying specific attention to integrating multi-omics datasets. Clinical queries and existing research pipelines are implemented in our framework and evaluated using declarative techniques specific to nested data processing.

8.2 Biomedical Data Sources

This section summarizes the data sources used in the TraNCE biomedical use cases. The majority of the datasets are provided from the Genomic Data Commons (GDC), which houses public datasets associated with the International Cancer Genome Consortium (ICGC). Some types are truncated for the sake of clarity, with full details available in the code repository [30].

Occurrences. An occurrence is a single, annotated mutation belonging to a single sample. The occurrence data source returns somatic mutations and associated annotation information for each sample. The `Occurrences` input is created by annotating each simple somatic mutation file (MAF) with the Variant Effect Predictor (VEP) [94]. VEP takes a distance flag to specify the upstream and downstream range from which to identify gene-based annotations. This flag is used to increase the flanking re-

gion of candidate genes associated to each somatic mutation for each sample. From a technical stand point, increasing the distance will increase the size of `candidates` and simultaneously increase the amount of skew. The type of `Occurrences` is:

$$Bag\,(\langle\, \texttt{sample} : \textit{string}, \texttt{contig} : \textit{string}, \texttt{start} : \textit{int}, \texttt{end} : \textit{int},\\ \texttt{reference} : \textit{string}, \texttt{alternate} : \textit{string}, \texttt{mutId} : \textit{string},\\ \texttt{candidates} : Bag\,(\langle\, \texttt{gene} : \textit{string}, \texttt{impact} : \textit{string}, \texttt{sift} : \textit{real}, \texttt{poly} : \textit{real},\\ \texttt{consequences} : Bag\,(\langle\, \texttt{conseq} : \textit{string} \rangle)\,\rangle)\,\rangle).$$

The attribute `candidates` identifies a collection of objects that contain attributes corresponding to the predicted effects a mutation has on a gene; i.e. *variant annotations* sourced from the Variant Effect Predictor (VEP) [94]. The `impact` attribute is a categorical value denoting the estimated consequence a mutation has to a gene based on sequence conservation. The `sift` and `poly` attributes provide additional impact scores determined from the Sift [95] and PolyPhen [96] prediction software. These scores estimate the influence a mutation has on functional changes to proteins based on amino acid substitution. The `consequences` for each candidate gene contain categorical assignments of mutation impact based on sequence ontology (SO) terms [97].

VEP provides a distance flag that specifies the upstream and downstream range used to identify gene-based annotations (i.e. the flanking region). This distance flag specifies the size of `candidates`, since more genes are assigned as candidates when a larger flanking region is specified. A larger value can be used to determine long-range functional connections.

Copy Number. Copy number variation (CNV) is the number of copies of a particular gene in a sample; thus, the copy number information is provided per gene. The copy number information is reported for each physical sample taken from a patient, denoted as `aliquot`. The type of the copy number information is:

$$Bag\,(\langle\, \texttt{aliquot} : \textit{string}, \texttt{gene} : \textit{string}, \texttt{cnum} : \textit{int} \rangle).$$

Variants. The `Variants` data source is based on the VariantContext [98] object that is used to represent variants from a Variant Call Format (VCF) file. This data structure represents one line, i.e. one variant, from a VCF file. Variants are identified by chromosome, position, reference and alternate alleles, and associated genotype information for every sample. We use an integer-based categorical assignment to genotype

calls to support analyses: 0 is homozygous reference with no mutated alleles, 1 is heterozygous with 1 mutated allele, and 2 is homozygous alternate with 2 mutated alleles. The type of `Variants` is:

$$\mathit{Bag}\,(\langle\,\texttt{contig} : \mathit{string}, \texttt{start} : \mathit{int}, \texttt{reference} : \mathit{string}, \texttt{alternate} : \mathit{string},$$
$$\texttt{genotypes} : \mathit{Bag}\,(\langle\,\texttt{sample} : \mathit{string}, \texttt{call} : \mathit{int}\rangle)\,\rangle).$$

Somatic mutations. Somatic mutations are stored in the GDC in Mutation Annotation Format (MAF), which is a flat datadump that includes a line for every mutation across all samples. The type of `Mutations` is:

$$\mathit{Bag}\,(\langle\,\texttt{sample} : \mathit{string}, \texttt{contig} : \mathit{string}, \texttt{start} : \mathit{int}, \texttt{end} : \mathit{int},$$
$$\texttt{reference} : \mathit{string}, \texttt{alternate} : \mathit{string}, \texttt{mutId} : \mathit{string}\,\rangle)$$

Variant Annotations. The variant annotations come from the VEP software, which takes as input mutation information either in VCF or MAF format and returns top-level mutation information augmented with two additional levels of gene and mutational impact information. The overall structure is similar to the `Occurrences` data source, except VEP returns a unique set of variant annotations that are not associated to a specific sample. The type of the `Annotations` data source is:

$$\mathit{Bag}\,(\langle\,\texttt{contig} : \mathit{string}, \texttt{start} : \mathit{int}, \texttt{end} : \mathit{int},$$
$$\texttt{reference} : \mathit{string}, \texttt{alternate} : \mathit{string}, \texttt{mutId} : \mathit{string},$$
$$\texttt{candidates} : \mathit{Bag}\,(\langle\,\texttt{gene} : \mathit{string}, \texttt{impact} : \mathit{string},$$
$$\texttt{sift} : \mathit{real}, \texttt{poly} : \mathit{real},$$
$$\texttt{consequences} : \mathit{Bag}\,(\langle\,\texttt{conseq} : \mathit{string}\rangle)\,\rangle)\,\rangle).$$

Protein-protein Interactions. Protein-protein interaction networks describe the relationship between proteins in a network. This `Network` input is derived from the STRING [99] database, which provides a likelihood score of two proteins interacting in a system. The network is represented with a top-level node tuple and a nested bag of edges, where each edge tuple contains an edge protein and a set of node-edge relationship measurements. The type of `Network` is:

$$\mathit{Bag}\,(\langle\,\texttt{nodeProtein} : \mathit{string}, \texttt{edges} :$$
$$\mathit{Bag}\,(\langle\,\texttt{edgeProtein} : \mathit{string}, \texttt{distance} : \mathit{int}\rangle)\,\rangle).$$

Gene Expression. Gene expression data is based on RNA sequencing data. Expression measurements are derived by counting the number of transcripts in an `aliquot` and comparing it to a reference count. The expression measurement is represented as Fragments Per Kilobase of transcript per Million mapped read (FPKM), which is a normalized count. The type of `GeneExpr` is:

$$Bag\left(\langle\, \texttt{aliquot} : string, \texttt{gene} : string, \texttt{fpkm} : real \rangle\right).$$

Pathway. In a simplified view, pathways are represented as a set of genes. Pathway information is downloaded as a list of curated gene sets from the Molecular Signatures Database (MSigDB) [100, 101]. The type of `Pathway` is:

$$Bag\left(\langle\, \texttt{pathway} : string, \texttt{genes} : Bag\left(\langle\, \texttt{gene} : string \rangle\right) \rangle\right).$$

Sample Metadata. The `Samples` input maps samples to their aliquots. For the sake of the use cases described in this section, `sample` maps to a patient and `aliquot` associates each biological sample taken from the patient. This dataset also contains additional clinical attributes that are associated to a patient. The type of `Samples` is:

$$Bag\left(\langle\, \texttt{sample} : string, \texttt{aliquot} : string, \texttt{tumorsite} : string, \texttt{toutcome} : string, \texttt{gleason} : int \rangle\right).$$

Sequence Ontology. The `SOImpact` input is a table derived from the sequence ontology [97] that maps a qualitative consequence to a quantitative consequence score (`conseq`). This is a continuous measurement from 0 to 1, with larger values representing more detrimental consequences. The type of `SOImpact` is:

$$Bag\left(\langle\, \texttt{conseq} : string, \texttt{value} : real \rangle\right).$$

Biomart Gene Map. The `Biomart` gene map input is exported from [102]. It is a map from gene identifiers to protein identifiers. This map is required to associate genes from `Occurrences` and `CopyNumber` to proteins that make up `Network`. The type of `Biomart` is:

$$Bag\left(\langle\, \texttt{gene} : string, \texttt{protein} : string \rangle\right).$$

Positional Gene Map. Gene mapping files provide the positional location of a gene on a genome, which is a combination of chromosome, start, and end position provided from a General Transfer Format (GTF) file. Each line of the GTF file maps a gene with its positional information [103]. The GTF file can be represented as a flat collection, `Genes`, with type:

$$Bag\,(\langle\, \texttt{gene} : \textit{string}, \texttt{description} : \textit{string}, \texttt{contig} : \textit{string},$$
$$\texttt{gid} : \textit{string}, \texttt{start} : \textit{int}, \texttt{end} : \textit{int}, \texttt{name} : \textit{string}\,\rangle).$$

8.3 Clinical Exploratory Queries

The identification of personalized diagnosis and treatment options is dependent on insights drawn from large-scale, multi-modal analysis of biomedical datasets. Practical clinical application of such targeted analyses require interfacing with electronic health record (EHR) systems, to provide a data processing environment that supports ease of integrating genomic, clinical, and other biomedical data linked to patients. For example, the Informatics for Integrating Biology and Beside (i2b2) [104] framework facilitates web-based cohort exploration, supporting selection and report generation on clinical attributes. Several proposed solutions for integrating genomic data into i2b2 have been proposed [105, 106, 107]. In these systems, genomic and clinical data are stored in separate databases and then combined in a backend plugin using the i2b2 API. A user first makes an analysis request from the interface. The request is sent to the backend application to retrieve and process the data sources in a distributed processing environment. The computed results are sent back to the interface for viewing. This section describes a set of clinical analysis requests, which are represented as TraNCE programs that operate on and return nested data for interactive exploration in a frontend. Figure 8.1 presents the programs, which build on each other to perform restructuring, integration, and aggregation operations. Each of the clinical programs is discussed below, followed by a discussion on their performance in the TraNCE framework.

Restructure Nested Data. The `OccurGrouped` program (Figure 8.1a) groups the somatic mutation occurrences in `Occurrences` by sample based on `Samples`, producing

```
OccurGrouped ⇐
for s in Samples union
 [{sample := s.sample, mutations :=
  for o in Occurrences union
   if s.sample == o.sample then
    [{mutId := o.mutId, ..., candidates :=
     for t in o.candidates union
      [{gene := t.gene, ..., consequences :=
       for c in t.consequences union
        for i in SOImpact union
         if c.conseq == i.conseq then
          [{conseq := i.conseq, score := i.value}]}]}]}]
```

(a) Restructure nested data: group datasets to return nested result.

```
OccurCNVJoin ⇐
for s in Samples union
 [{sample := s.sample, mutations :=
  for o in Occurrences union
   if s.sample == o.sample then
    [{mutId := o.mutId, ..., candidates :=
     for t in o.candidates union
      for g in CopyNumber union
       if g.gene == t.gene && g.sample == o.sample then
        [{gene := t.gene, cnum := g.cnum, ...,
          consequences :=
           for c in t.consequences union
            for i in SOImpact union
             if c.conseq == i.conseq then
              [{conseq := i.conseq, score := i.value}]}]}]}]
```

(b) Integrate at nested level: associate datasets to return nested result.

```
OccurCNVAgg ⇐
for s in Samples union
 [{sample := s.sample, mutations :=
  for o in Occurrences union
   if s.sample == o.sample then
    [{mutId := o.mutId, ..., candidates :=
     sumBy^score_gene(
      for t in o.candidates union
       for g in CopyNumber union
        if g.gene == t.gene && g.sample == o.sample then
         for c in t.consequences union
          for i in SOImpact union
           if c.conseq == i.conseq then
            [{gene := t.gene,
              score := t.impact * g.cnum * i.value}])}]}]
```

(c) Aggregate at nested level: associate datasets, aggregate, and return nested result.

Figure 8.1: The NRC$_{agg}$ programs for clinical exploration. Ellipses represent additional attributes from `Occurrences` and `CopyNumber`.

a collection of nested mutation information for each sample. The grouping involves the whole of the mutations and the majority of the attributes are persisted in the output. The program also associates a quantitative value to the consequences at the lowest level in the process, as seen previously in the `HybridMatrix` program from the driver gene analysis. The result of the program could be consumed by a web-interface that provided a detailed view of annotated mutations across a cohort of patients.

Integrate Data at Nested Level. The `OccurCNVJoin` program (Figure 8.1b) extends `OccurGrouped` by associating copy number information (`CopyNumber`) to each of the genes in the `candidates` collection for each mutation in `Occurrences`. The result is returned group by sample, and the majority of the fields from `Occurrences` are persisted in the output. This program addresses the situation where additional biomedical datasets are integrated for exploration in a consolidated view, without any aggregation.

Aggregate Data at Nested Level. The final clinical program `OccurCNVAgg` (Figure 8.1c) combines all aspects of the first two programs and adds an additional aggregation. Mutations are associated to copy number data and an aggregate value is created with the nested mutational impact attribute from `consequences` in `Occurrences`. The scores are returned for each candidate gene within each mutation, and the final output is grouped by sample. This is similar to the `HybridMuts` running example, but with additional fields returned.

8.3.1 Runtime performance

The clinical programs were executed following the same Spark configuration as in Section 7.2. The full TCGA [80] dataset (Pancancer) and the TCGA breast cancer dataset (BRCA) are used. The pancancer datasets uses 42GB of `Occurrences` with a 10000 base flanking region and 34GB of `CopyNumber`. The breast cancer dataset is 168 Megabytes (MB) of `Occurrences` with 10000 base flanking region and 4GB of `CopyNumber`. The runtime of a program is measured by first caching all inputs in memory.

The runtimes for each clinical program are presented in Figure 8.2 for the standard and shredded compilation routes. The time for unshredding is also provided. The

smaller breast cancer dataset shows performance benefits of the shredded representation over flattening methods. The restructuring in `OccurGrouped` is 80× faster when returning shredded output and 12× as performant when returning nested output. There is up to a 9× performance gain for the integration and aggregation programs. The results show that the shredded compilation route can bring advantages even for small-scale datasets regardless of how the output type is returned. For the larger sample set, the flattening methods are unable to scale at all, overloading the available memory on the system during each program execution.

The results here confirm that the results from Chapter 7 are relevant to real-world datasets. The clinical programs are classified as nested-to-nested with wide tuples, which tend to be the worst case for flattening methods. Further, the additional operations on lower-level dictionaries display how the cost of unshredding can be reduced. For example, the `OccurGrouped` step is very cheap, but becomes more expensive when nested output is reconstructed (Unshred). As additional operations are added the cost of unshredding decreases about 3×. The cost of shredding does increase but at a slower rate, incurring only a portion of the cost. Overall, the results confirm that flattening methods are unable to scale for increasing sample sizes and the shredded representation could be of benefit.

8.4 Multi-Omics Driver Gene Identification

Cancer progression can be determined by the accumulation of mutations and other genomic aberrations within a sample [108]. *Somatic mutations* are cancer-specific mutations that are identified within each sample, and are identified by comparing a sample's cancerous genome to a non-cancerous, reference genome. Note that the term mutation is often used interchangeably with variant. *Candidate genes* are assigned to mutations based on the proximity of a mutation to a gene. In a naive assignment, candidacy is established if the mutation lies directly on a gene; however, mutations have been shown to form long-range functional connections with genes [109], so candidacy can best be assigned based on a larger flanking region of the genome.

Somatic mutations that play a driving role in cancer often occur at low frequency, making cohort analysis across many samples important in their identification. Fur-

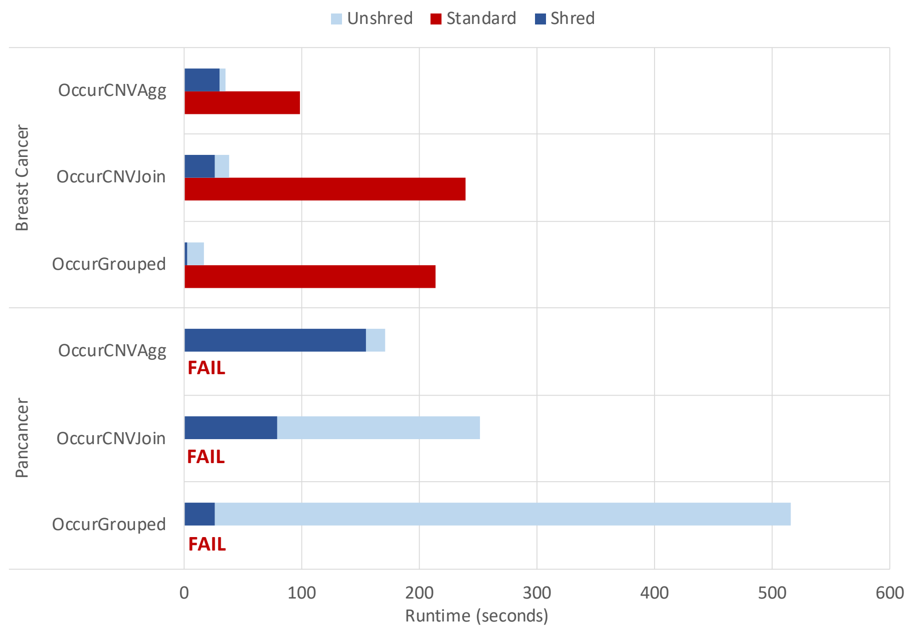


Figure 8.2: Results for the clinical exploration programs. The standard compilation route fails for all runs with the Pancancer dataset.

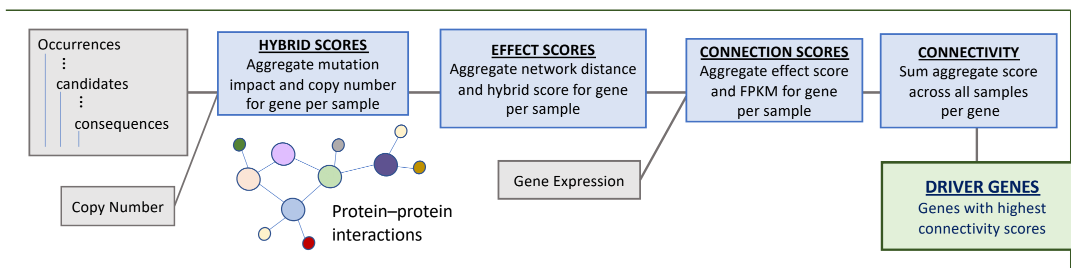


Figure 8.3: Summary of the cancer driver gene analysis. The pipeline starts by integrating somatic mutations and copy number variation, further integrates network information, and gene expression data. The genes with the highest connectivity scores are taken to be drivers.

ther, a cancer profile is more than just a consequence of a single mutation on a single gene. Gene interactions, the number of such genes, and their expression levels can provide a more thorough look at cancer progression [108]. This use case focuses on such a multi-omics analysis, which defines a set of programs that integrate annotated somatic mutation information (`Occurrences`), copy number variation (`CopyNumber`), protein-protein network (`Network`), and gene expression (`GeneExpr`) data to identify driver genes in cancer [73]. This analysis provides an integrated look at the impact cancer has on the underlying biological system and takes into account the effects a mutation has on a gene, the accumulation of genes with respect to both copy number and expression, and the interaction of genes within the system.

Figure 8.3 provides an overview of the cancer driver gene analysis. The pipeline starts with the integration of mutation and copy number variation to produce a set of hybrid scores for each sample. The hybrid scores are then combined with protein-protein network interactions to determine effect scores. The effect scores are further combined with gene expression information to determine the connection scores for each sample. The analysis concludes by combining the connection scores across all samples, returning connectivity scores for each gene. The genes with the highest connectivity scores are considered drivers. Figure 8.4 programs of the driver gene analysis work in pipeline fashion, where the materialized output from one program is used as input to another later on in the pipeline. The programs of this analysis are summarized below, followed by performance metrics for SparkSQL as well as the standard and shredded compilation routes.

Hybrid Scores. The hybrid score program `HybridMatrix` (Figure 8.4a) is the first step in the pipeline. `Samples` provides a map between `sample` and `aliquot` used to join `CopyNumber`, and the hybrid scores are then determined for every `aliquot`. In addition, conditionals are used to assign qualitative scores based on the human-interpretable level of impact (`impact`). The `SOImpact` information is used to integrate values from the nested `consequences` collection into the hybrid score.

The program persists `aliquot` in order to associate more genomic measurements related to that aliquot later in the pipeline. These hybrid scores provide a likelihood score of a gene being a driver within a specific aliquot based on both accumulated impact of somatic mutations and copy number variation. The analysis continues to integrate further information to increase the confidence of driver gene scores.

```
HybridMatrix ⇐
 for s in Samples union
  [{sample := s.sample, aliquot := s.aliquot, scores := sumBy^score_gene(
    for o in Occurrences union if o.sample == s.sample then
     for t in o.transcripts union
      for n in CopyNumber union if s.aliquot == n.aliquot &&  n.gene == t.gene then
       for c in t.consequences union
        for v in SOImpact union if c.conseq == v.conseq then
         [{gene := t.gene, ... score := let impact :=
           if t.impact == "HIGH" then 0.8
           else if t.impact == "MODERATE" then 0.5
           else if t.impact == "LOW" then 0.3
           else if t.impact == "MODIFIER" then 0.15
           else 0.01 in impact * v.value * (n.cnum + .01) * sift * poly}])}]
```

(a) Hybrid score calculation: combined somatic mutation and copy number variation.

```
SampleNetwork ⇐
for h in HybridMatrix union
 [{sample := h.sample, aliquot := h.aliquot, nodes := sumBy^score_gene(
   for g in h.scores union
    for b in Biomart union if g.gene == b.gene then
      for n in Network union
       for e in n.edges union if e.edgeProtein == b.protein then
         [{nodeProtein := n.nodeProtein, score := e.distance * h.score}])}]
```

(b) By-sample networks: combined hybrid score and protein-protein interactions.

```
EffectMatrix ⇐
for h in HybridMatrix union
 [{sample := h.sample, aliquot := h.aliquot, scores :=
  for s in SampleNetwork union if h.sample == s.sample && h.aliquot == s.aliquot then
    for n in s.nodes union
     for b in Biomart union if n.nodeProtein == b.protein then
       for y in h.scores union if y.gene == b.gene then
         [{gene := y.gene, score := n.score * h.score}]}]
```

(c) Effect score calculation: by-sample networks aggregated by gene.

```
ConnectMatrix ⇐
for e in EffectMatrix union
 [{sample := e.sample, aliquot := e.aliquot, scores := sumBy^score_gene(
   for e in s.scores union
    for g in GeneExpr union if e.gene == g.gene then
      [{gene := e.gene, score := e.score * g.fpkm}])}]
```

(d) Connection score calculation: combined effect score and gene expression data.

```
Connectivity ⇐
sumBy^score_gene(
for s in ConnectMatrix union
 for c in s.scores union
  [{gene := c.gene, score := s.score}])
```

(e) Gene connectivity: connection scores summed across all samples for each gene.

Figure 8.4: The NRC_{agg} program of the driver gene analysis.

By-Sample Networks. The second step in the pipeline `SampleNetwork` (Figure 8.4b) builds individual aggregated networks for each (`sample`, `aliquot`) pair in the materialized output of `HybridMatrix`. For each sample, the product of the `score` and edge protein `distance` for each edge in the network is taken. Genes are associated to proteins based on the mapping provided in the `Biomart` gene map table. The sum aggregate of these values is then taken for each node protein in `Network`, while maintaining top-level sample groups.

The `SampleNetwork` program produces an intermediate score for each protein in the network by weighting the hybrid scores of nearby proteins in the network (edges) based on their distance scores; thus, this is an intermediate aggregation of the network data with the hybrid scores using only the edges in the network.

Effect Scores. To complete the integration of network data with the hybrid scores, the next step is to integrate the nodes in the `Network` to produce effect scores. Effect scores are produced by combining the accumulated edge-based hybrid scores from `SampleNetwork` with the hybrid score for each protein node for each sample in the materialized output of `HybridMatrix`. As in `SampleNetwork`, genes are associated to proteins using the `Biomart` mapping table. The effect score is another likelihood measurement for a gene being a driver gene for cancer.

Connection Scores. The `ConnectMatrix` program calculates the connection scores. A connection score is the product of the effect score and the FPKM value from the `GeneExpr` table. Gene expression data is combined with the materialized output of `EffectMatrix` to determine the connection scores for each gene within every sample. Given the pipeline nature of these queries, the connection scores for each gene are the accumulated somatic mutation, copy number, protein-protein network, and gene expression data for each sample. The connect score can be used to determine the likelihood of a gene being a driver in a specific sample. In theory, this likelihood measurement should have more confidence than the hybrid or effect scores.

Gene Connectivity. At this point in the analysis, all the genomic measurements have been integrated to produce high-confidence likelihood connection scores for each gene within each sample. The final step is to combine across all samples to identify the highest scoring genes over all samples; this is the gene connectivity. Gene connectivity uses the materialized output of `ConnectMatrix`, summing up the connec-

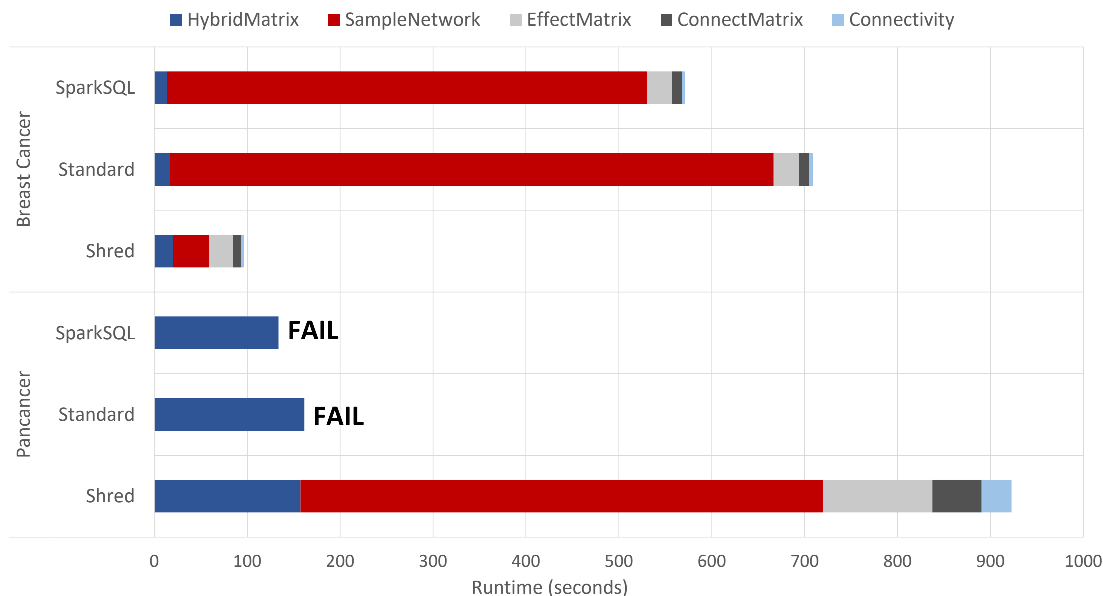


Figure 8.5: Runtimes for each of the stages of the driver gene analysis. The standard compilation fails at the `SampleNetwork` program.

tion scores for each gene across all samples. The genes with the highest connection scores are taken to be drivers. The final output of of `Connectivity` is sorted and the top genes are investigated as likely driver genes for cancer. Further confidence can be gained by fine-mapping techniques [110].

8.4.1 Runtime performance

The driver gene analysis was executed following the same Spark configuration as in Section 7.2. The full TCGA [80] dataset (Pancancer) and the TCGA breast cancer dataset (BRCA) are used. The pancancer dataset uses 280GB of `Occurrences` [79, 94], 4GB of `Network` [99], 23G of `GeneExpr`, and 34GB of `CopyNumber` (34GB). The breast cancer dataset uses the same network information, 6GB of `Occurrences`, 2GB of `GeneExpr`, and 4GB of `CopyNumber`. The runtime of a program is measured by first caching all inputs in memory.

Figure 8.5 shows the runtimes for SparkSQL, standard (STANDARD), and shredded

(SHRED) compilation for the breast and pancancer datasets. The final output is flat, so there is no unshredding costs consider. Since there are about 10× more samples in the pancancer dataset, this experiment looks at how the methods scale to an increasing number of samples. The `HybridMatrix` program involves a nested-to-flat operation on the `Occurrences` data source, which results in comparable performance across all methods for both size datasets. The methods diverge significantly at `SampleNetwork`, which performs a nested join to combine hybrid scores and network information that leads to an explosion in the amount of shuffled data. This is an expensive operation for flattening methods even when the dataset is small. For the larger dataset, the flattening methods (SparkSQL, STANDARD) produce an intermediate join result with 16 billion tuples and shuffles up to 2.1TB before crashing. SHRED reduces the size to 10 billion tuples and shuffle to 470GB. The shredded representation provides a 7x speed up for the smaller breast cancer dataset and is able to run to completion for all stages when flattening methods are unable to complete at all.

This analysis follows the workflow of [73], which terminates with an identification of driver genes. The three top driver genes reported from our analysis were TP53, FLNA, and CSDE1. All these genes have previously been reported as important for their role in cancer. Future work should explore the pancancer results of this analysis, potentially comparing tumor-site specific driver genes to the identified pancancer driver genes.

8.5 Mutational Burden Predictions

High mutational burden can be used as a confidence biomarker for cancer therapy [111, 112]. One key measure is the total number of somatic mutations in a tumor sample, or *tumor mutational burden* (TMB). Mutational burden can be used directly as a likelihood measurement for immunotherapy response [111], similar to the likelihood scores in the driver gene pipeline. Burdens can also be used as features for a classification problem. The progression of some cancers could make it impossible for a clinician to identify the tumor of origin [113]. The ability to classify tumor of origin from a cohort of cancer types can be clinically actionable, providing insights into the diagnosis and type of treatment the patient should receive. Programs to build burden-based feature vectors are considered for predicting tumor of origin in a pancancer dataset.

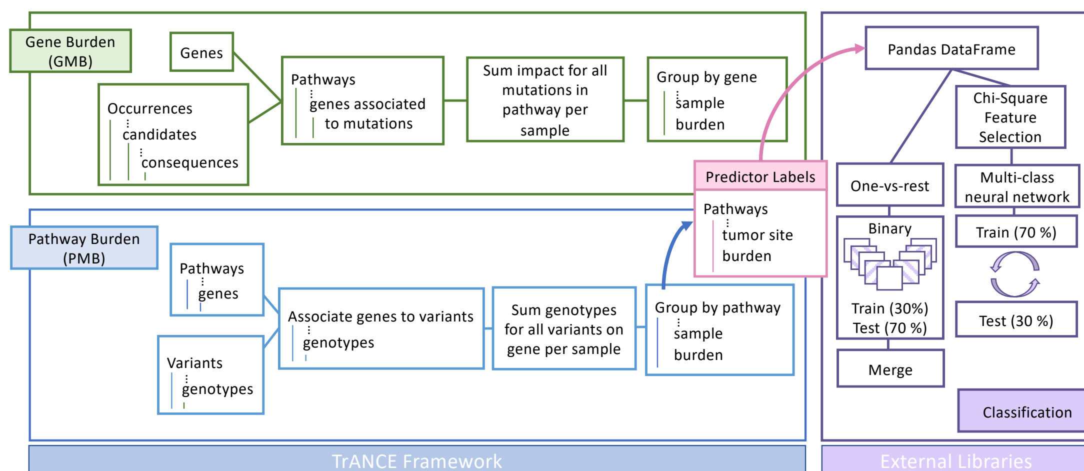


Figure 8.6: Workflow diagram representing the burden-based analyses for both genes and pathways, and downstream classification problem. The results of the pathway burden analysis feed into a classification analysis using multi-class and one-vs-rest methods to predict tumor of origin.

The burden-based analysis of this use case focus on two subcalculations of TMB: gene mutational burden (GMB) and pathway mutational burden (PMB). GMB is the total number of somatic mutations present in a given gene per tumor sample. PMB is the total number of somatic mutations present in a given pathway per tumor sample. Figure 8.6 summarizes these two programs and the downstream classification analysis in an external Python environment. Since a pathway is a set of genes, GMB is a partial aggregate of pathway burden; ie. PMB is the sum of all the gene burdens for each gene belonging to a pathway. The gene burden program is then shown using `Occurrences` and pathway burden using `Variants`. Each of the programs persists a predictor label `tumorsite` from `Samples`. The burden-based programs are described in detail below, along with the downstream learning application and associated performance results.

Gene Burden. The `GeneBurden` program (Figure 8.7a) uses annotations within `Occurrences` to determine gene association. When a candidate gene set is created based on a large flanking region, the gene burdens could be over-estimated. The program could use impact information instead of raw count to measure the mutational burden of a gene.

```
GeneBurden ⇐
for g in Genes union
 [{gene := g.gene, burdens :=
  sumBy^score_sample(
   for o in Occurrences union
    for s in Samples unionif o.sample == s.sample then
      for t in o.candidates unionif g.gene == t.gene then
        [{sample := o.sample, tumorsite := t.tumorsite, burden := 1.0}])}]
```

(a) Gene burden program: number of mutations on each gene per sample.

```
PMB ⇐
for p in Pathway union
 [{pathway := p.pathway, burdens :=
  sumBy^burden_sample(
   for v in Variants union
    for g in p.genes union
     for m in Genes union if g.gene == m.gene then
      if v.contig == m.contig && v.start >= m.start && v.end <= m.end then
       for c in v.genotypes union
        for s in Samples unionif c.sample == s.sample then
        [{sample := c.sample, tumorsite := s.tumorsite, burden := c.call}])}]
```

(b) Pathway burden program: number of alternate alleles on each gene per sample.

Figure 8.7: The burden-based programs for feature vector construction.

Pathway Burden. The `PMB` program (Figure 8.7b) performs a VCF-based analysis using the `Variants` data source. The program first iterates `Pathway` creating a top-level pathway group, and then performs a sum-aggregate of the nested genotype calls for each sample corresponding to that gene. Variants are associated to a gene if it lies within the mapped position on the genome, which is a more strict association than designated by the candidate gene set association in `GeneBurden` above. This program could be altered to include a larger flanking region by changing the equalities on `start` and `end` to use a range.

8.5.1 Classification with burden-based features

The materialized results of the above programs are passed as feature vectors to a learning classifier. Classification of tumor origin has been previously explored with various cancer biomarkers [114, 115, 116, 117]. The goal of the classification problem is to identify tissue of origin from the whole TCGA dataset using pathway burden features based on raw mutation count.

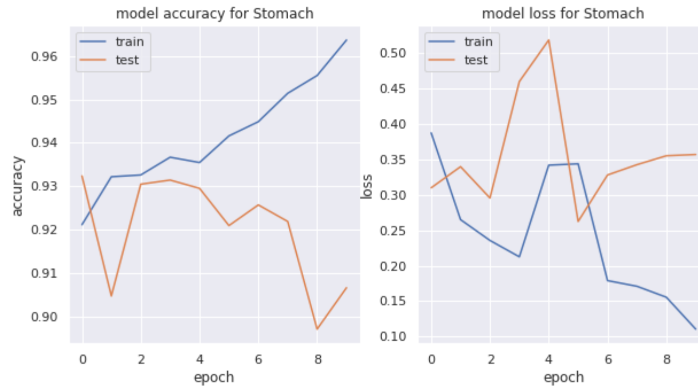
In order to interface with external machine learning libraries, the burden-based pro-

Figure 8.8: Accuracy and loss of the multi-class neural network for tumor tissue site.

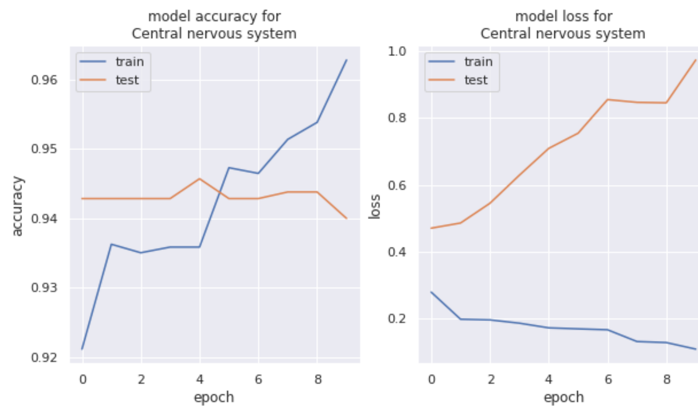
grams are compiled into Zeppelin notebooks where the output is available once the program is executed. Learning procedures can then be applied directly in Spark/Scala, or the ZeppelinContext can be used to read a Spark DataFrame as a Pandas DataFrame. For this example, focus is on the Pandas representation to highlight how a user can interact with outputs from an NRC_{agg} program using their external library of choice. Given this data processing pipeline, TraNCE is used for the heavy lifting portion that integrates and restructures the datasets to produce feature matrices; the assumption being that the data is reduced to a size reasonable enough for the in-memory processing of standard statistical libraries.

Once represented as a Pandas DataFrame, the data is split for training and testing using scikit-learn and neural networks are constructed with keras. For the whole of the TCGA dataset, we use a minimum cut-off of 200 representative samples for any given tissue site. This leaves nine different tumor tissue sites available for classification: breast, central nervous system, colon, endometrial, head and neck, kidney, lung, ovary, and stomach.

The model is trained using a fully-connected, feed-forward multi-class neural network for tumor tissue site, using 1600 pathways selected by the Chi-squared test as the features. The neural network uses LeakyReLu [118] with $alpha = 0.05$ as the activation function, and dropout layers [119] with $dropout_rate = 0.3$ after each fully connected



(a) Stomach



(b) Central nervous system

(c) Head and neck

Figure 8.9: Accuracy and loss for the tumor tissue site based binary network, includes results for the three worst-performing classes from the multi-class network.

layer (dense layer) before the output. This model is trained using a categorical cross-entropy loss function and an Adam optimizer [120]. The network has a Softmax output, which can be interpreted as a probability distribution over nine different tumor tissue sites. The data is randomly split into two folds, 70% for training and 30% for testing.

Next, the previous method is extended via the "one-vs-rest" method [121], which decomposes a multi-classification problem into multiple binary classification problems and each binary classifier is trained independently. For every sample, only the most "confident" model is selected to make the prediction.

Each binary classifier is a fully connected, feed-forward neural network, using all 2230 pathways as the features. These are set up the same as the multi-class networks, except with dropout layer *dropout_rate = 0.15* and a binary cross-entropy loss function. The binary networks have Sigmoid output, which can be interpreted as a probability of a certain type of tumor tissue site corresponding to this model. For each model, the data is randomly split into two folds as with the tumor-site network.

Nine independent binary classifiers are trained for each type of tumor tissue site. These binary models predict the likelihood that the given pathway burden measurements of a patient are associated with the tumor site represented by that model. After training each binary model, predictions are made using the entire dataset, and the computed results are merged. The probabilities from all models are compared for each patient from the testing dataset, classifying the patient according to the highest likelihood. For example, suppose there are two models, a breast model that predicts a breast-site likelihood of 0.8 and a lung model that predicts a lung-site likelihood of 0.6 for the same patient. The system compares these two probabilities and classifies tumor of origin as breast.

Difference in sampling procedures aside, the multi-classifier and the binary models in the one-vs-rest method have one key difference. When using pathway burden features, pathways that are highly correlated with a specific tumor site could be overpowered by pathways that show strong signal for cancer in general. The multi-classifier could compromise features specific to tumor of origin in an attempt to achieve best performance overall. This can lead to particularly inaccurate results when the data distribution is uneven. The binary models are eager to select the best feature weights for the repre-

sentative tumor of origin, providing more opportunities for tumor-specific features to stand out.

Multi-classification performance. Figure 8.8 shows the accuracy and loss of the multi-class neural network for tumor tissue site for 30 epochs. The overall accuracy is 42.32%, calculated from the confusion matrix by adding all 444 correctly predicted labels together and dividing by the 1049 testing samples. Most misclassifications were predicted to be breast cancer, likely attributed to the data imbalance problem of the training dataset. An imbalanced data distribution forces a model to learn features corresponding to highly populated labels, reducing training loss while skewing overall prediction performance.

Different types of cancer may not contain enough dominant features for a simple multi-class model to distinguish differences among tumor origin site. Even pathways that play a key role in any cancer, such as pathways specific to disruption in cell-cycle, could not be providing enough signal to act as a determinant for cancer types. This could be because other pathways are washing out the signal of more important pathways, or it could mean that pathway burden alone is not providing the whole story.

One-vs-rest classification performance. Figure 8.9 displays the accuracy and loss of three binary networks for 10 epochs. The three worst-performing classes from the multi-class network: stomach, head and neck, and central nervous system are presented, which all resulted in testing accuracies above 90% in the one-vs-rest method. The accuracy and loss of the other binary models are provided in the Appendix A.3. The combined accuracy of all binary models is 78.44%, calculated as the correctly predicted labels (2744) divided by total samples (3498). Overall performance of the one-vs-rest method is far better than the multi-classifier performance.

Future Exploration. Further exploration into the pathway signal profiles of each tumor site could be considered for future work. Gene burden performance could be compared to the performance of pathway burden in order to identify genes that are the main drivers for pathway signal. The identification of predominant pathways and genes for certain tumor sites could provide insight into specific cancer profiles and determine the overall confidence of using burden-based features for tumor site classification. Additional multi-class problems in this domain could consider integrating other features, such as additional genomic measurements. Most of this is taken into consid-

eration in the Chapter 9, which transforms the classification analysis into UDFs that are leveraged with the programs.

8.6 Additional TraNCE Performance Results

This section presents additional results related to TraNCE performance for biomedical applications. The experiments in Chapter 7 evaluated performance on both statically-sized simulated data and a statically-sized cluster, and only focused on the sampling technique for skew-handling. This section will look at the scalability of TraNCE for increasing number of top-level records and increasing amounts of compute resources (Section 8.6.1). All of the heavy key identification methods: full, partial, sample, and slice are then explored using the biomedical datasets, which includes an assessment of the skew-handling methods for real-world datasets (Section 8.6.2). The section concludes with a note on how sharing can improve the performance of the shredded representation in Section 8.7.

8.6.1 Scalability

This section uses programs from the driver gene and burden-based analyses to illustrate the scalability benefits of the platform. The burden-based programs are used to assess scalability of both compilation routes for increasing data size and constant cluster size. The driver gene programs `HybridMatrix` and `SampleNetwork` are then used to measure scalability for constant data size and increasing cluster size for the shredded compilation route only.

Increasing top-level tuples. To assess the scalability of TraNCE with respect to data size, the burden-based programs are presented for the standard and shredded compilation routes for an increasing number of top-level records. The experiment was run on Spark 2.4.2 (Scala 2.12 / Hadoop 2.7) with one worker, 10 executors, 2 cores and 20 Gigabyte (GB) memory per executor, and 16GB of driver memory. Due to resource limitations of this cluster, the experiment uses an 11.2GB `Variants` dataset from Chromosome 22 of Phase 3 1000 Genomes[78, 122], which is 2504 samples total.

Figure 8.10 displays the standard and shredded compilation runtimes for an increas-

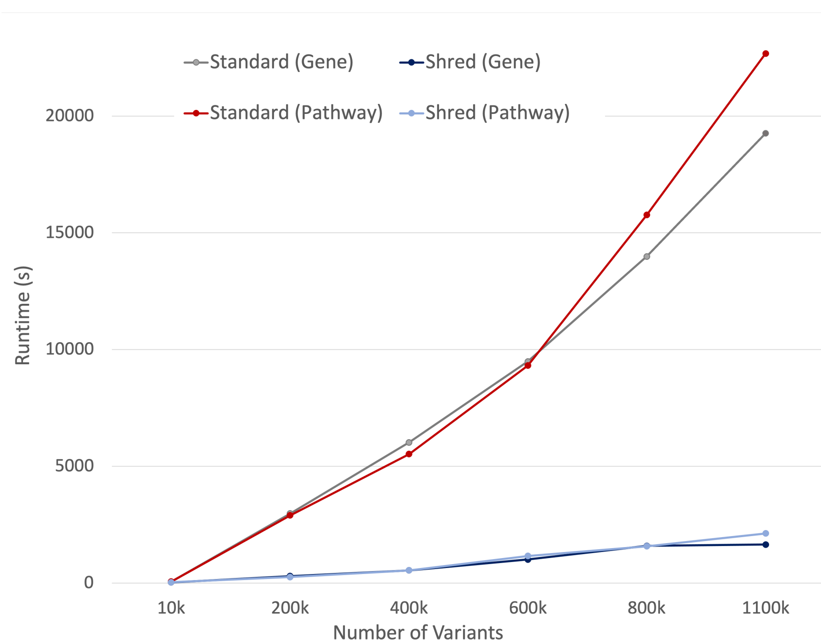


Figure 8.10: Runtime burden-based analyses for an increasing number of top-level variants.

ing number of variants, using both gene and pathway burden. The results show that the flattening methods of the standard route have a greater increase in runtime as the number of variants increase compared to the shredded route. After 600-thousand variants the standard pathway burden run increases at a greater rate than the corresponding gene burden run. There are two main advantages of the shredded route here. First, the shredded representation avoids carrying around extra data, such as the genotypes information when `Variants` are joined with `Genes`. Second, the result of flattening `Variants` results in a large intermediate. The whole file contains roughly 1103600 variants and more than 2500 samples, which produces a result with over 2.7 billion items. The performance benefits exhibited in this experiment are only for a single chromosome; as such, distributed computing becomes even more of a necessity when processing whole genomes or considering more samples. These results highlight the advantage of using the shredded representation even for the shallow nesting of the VariantContext structure.

Increasing cluster size. This experiment uses the driver gene programs to assess scalability of the shredded compilation route as the amount of compute resources

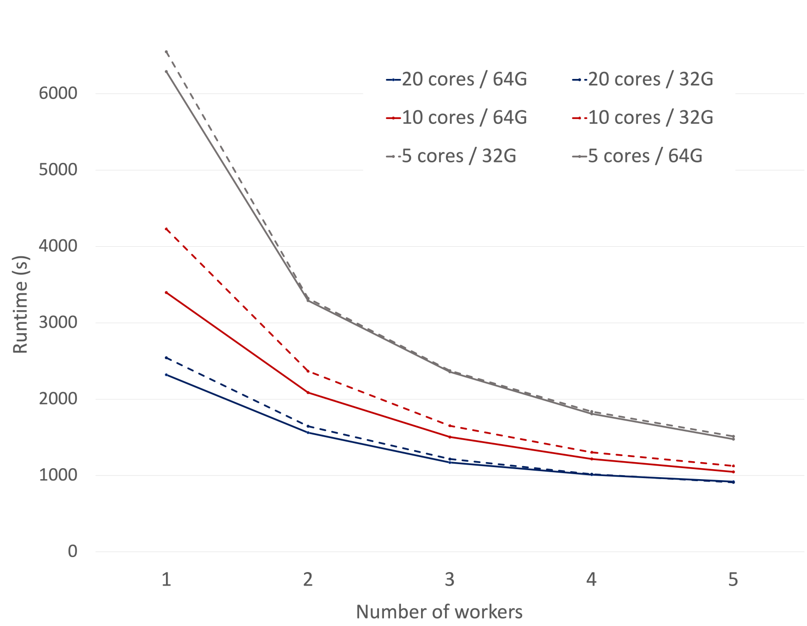


Figure 8.11: Scalability for the driver gene analysis measured using `HybridMatrix` and `SampleNetwork` programs produced from the shredded compilation route for a variety of cluster configurations.

increase. Figure 8.11 displays the combined runtime for the first two, most expensive steps of the driver gene analysis - `HybridMatrix` and `SampleNetwork`- for an increasing amount of workers and a variety of cluster configurations. The programs are run with the pancancer datasets, including 280GB of `Occurrences` [79, 94], 4GB of `Network` [99], and 34GB of `CopyNumber` (34GB). Focus is on the shredded compilation route, since the standard compilation route was unable to perform for this scale of data.

An increasing number of cores (5, 10, 20) per worker to evaluate performance based on the number of processing units available to each worker. For each of these runs 32GB and 64GB are provided per worker to assess how available memory impacts performance. For each run workers are added with constant amounts of resources; for example, for the 5 cores and 32G run, the one worker mark represents 5 cores and 32G total, whereas the two worker mark represents 10 cores and 64G total. The number of cores and number of workers are used collectively to measure overall distribution.

The single worker runs exhibit the largest variation in runtime. While the single

worker with 10 cores completes 14 minutes faster when more memory is available, the runtimes for the single workers with 5 and 20 cores are less impacted when memory increases. In general, the total number of cores and the amount of memory available to each working unit is more important when there are fewer workers available to a cluster. All runs converge to a point where the shuffling overhead dominates the total execution time and what is left is not parallelizable. This shuffling overhead is related to the expense of the `SampleNetwork` program, which requires a significant amount of shuffling to perform a join on nested attributes - previous results (Figure 8.5) reported 470G of shuffled data for the shredded compilation route. Even in a situation where large amounts of shuffling cannot be avoided, the addition of five workers has saved 24 - 78 minutes depending on the amount of resources available to each worker. These scalability results highlight how adding cluster resources will improve the performance of current data pipelines.

8.6.2 Skew-handling

This experiment evaluates the skew-handling methods in biomedical applications. The goal is to look at skew-handling methods for increasing amounts of skew that arises naturally from biomedical datasets, and assess each of the heavy key identification methods mentioned in Section 5.1.1. The `HybridMatrix` grouped by tumor site and related programs that group by gene, pathway, and gene family are explored with and without skew-handling for both the standard and shredded compilation routes. The grouping of mutations by gene and pathway are represented by the `GeneBurden` and `PMB` programs from Application 1. Gene families are a special instance of pathway, so this grouping also follows `PMB`. The categories include 92 tumor sites represented in the TCGA dataset, 58-thousand genes from a gene map file, 2230 pathways based on curated gene sets from The Molecular Signatures Database (MSigDB) [101, 123], and the 8 gene family classifications also from MSigDB. A subset of the `Occurrences` dataset (12.3G) is used, which represents 10% of the full dataset. Each program was run on a cluster with 5 workers each with 20 cores and 16G. Figure 8.12 displays the runtimes for each compilation route for the full, partial, sample, and slice heavy key identification techniques.

Performance for increasing amounts of skew. The tumor site and gene groupings exhibit low amounts of skew; all methods are able to run to completion with the skew-aware techniques only adding a slight overhead related to the cost of performing the heavy key calculation. While there are 92 tumor sites, the join operation for `tumorsite` is at top-level while all the others are on a nested `gene` attribute. The gene-based operation can partition by 58-thousand genes, producing enough distribution to not cause a bottleneck. The data imbalances exhibited for the tumor site and gene grouping are not enough to completely overwhelm any resources on a single node, though the standard route does show burdened runtimes. For these low skew programs, the standard compilation runs exhibit 2.5× overhead for tumor site on average and 80× for gene in comparison to the shredded variant. The standard compilation route was unable to perform at all for higher amounts of skew regardless of skew-handling techniques. This highlights the natural skew-resilience of the shredded representation, which is not burdened by storing the nested collections locally.

The pathway run exhibits moderate amounts of skew. Here, the skew-unaware method is unable to perform at all and spills 540G of data to disk before crashing. This result is expected with increased amounts of skew since the values associated with heavy keys will completely overwhelm the resources on each node. The gene family run exhibits high amounts of skew. Again, the skew-unaware method is unable to perform at all. The skew-aware methods for pathway and gene family highlight the performance gains of avoiding the key-based partitioning strategy for heavy keys. The performance gain increases for higher amounts of skew since the majority of the values will be associated to heavy keys and these values will remain stationary during the skew-aware operation.

Best performing heavy key identification method. The slice procedure exhibits interesting behavior when estimating heavy keys in large intermediates. When skew is low and the shredded representation is used, the slice procedure is the most expensive, whereas slice is the best performing for the standard route for low skew. In the standard procedure, `Occurrences` is flattened before joining with the gene table that will instantiate the nested mutation groups. This flattening produces more values for the skew-handling strategy to process, which is where the slice procedure performs better. On the other hand, the shredded representation runs the estimate on the lighterweight, first-level source of `Occurrences` and ends up over estimating the heavy keys.

Overestimating heavy keys can lead to longer processing times due to broadcasting larger intermediates. For the moderate and high amounts of skew in pathway and gene family groupings, the shredded representation performs the best with the slice method. For these programs, full, partial, and sample have longer processing times due to over estimation.

The slice procedure is the best performing of all methods, providing a quick estimate for heavy keys while keeping the amount of broadcasted data low. Given that the full procedure should accurately identify all heavy keys, this suggests that it is advantageous to only apply skew-aware operations to the most immediately identifiable heavy keys than it is to fully estimate heavy keys. This contradicts the results from the simulated dataset (Section 7.4.1). The skew operations in the TPC-H dataset are on database-key columns, whereas the biomedical dataset performs key-based operations on diverse datasets that do not have explicit functional dependencies. This makes the "keys" of the biomedical dataset more naturally distributed and easier to estimate based off the first section of data. The TPC-H dataset is more uniform and requires more thorough probing to determine the outliers. These results suggest that slice procedures are best for datasets without key information, while sampling is better when keys are known.

8.7 Sharing in the shredded representation

The use cases of this section use an `Occurrences` input based on the occurrences endpoint of the ICGC data portal [79], which returns JSON-formatted data following the structure of `Occurrences`. In this representation, annotations are repeated within the nested `candidates` collection for mutations that are shared across samples. This sharing can be exploited to create an even more succinct shredded representation of the `Occurrences` data source. The goal of this experiment is to measure the amount of sharing that can be leveraged in the shredded representation using the biomedical datasets.

The following `BuildOccur`program can construct an `Occurrences` data source, with all somatic mutations in the `Mutations` data source and all associated unique annotations in the `Annotations` data source.

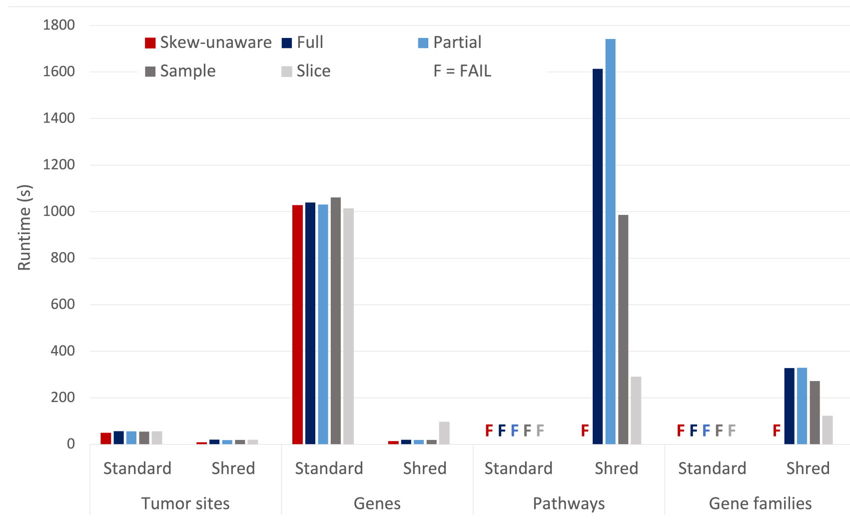


Figure 8.12: Performance comparison of the skew-handling techniques for both the standard and shredded compilation routes. Queries are organized based on increasing amounts of skew, such that tumor sites is representative of low skew and gene families of high skew.

```
BuildOccur ⇐
for m in Mutations union
 for v in Annotations union
  if m.contig == v.contig && m.start == v.start
     && m.alternate == v.alternate then
   [{sample := m.sample, mutId := v.mutId, ...,
     candidates := v.candidates}]}]
```

To explore the benefits of sharing, the BuildOccurprogram is executed for both the standard and shredded compilation routes. One somatic mutation (MAF) file from the breast cancer dataset (Mutations) is used containing 120988 tuples, and the associated unique set of 58121 VEP annotations (Annotations). When the somatic mutations are joined with annotations on (contig, start, alternate) in the standard route, the result contains 5170132 tuples nested within the candidates collections of the whole output. For the shredded route, the somatic mutations are joined with the top-level source Annotations_top, which has replaced the candidates values with labels. The first-level output BuildOccur_cands is the same as the input Annotations_candidates, which has 3777092 tuples. The shredded representation reduces the total size of the transcripts by over one million tuples.

The results of this experiment are based off a small dataset. Since many of the samples will share mutations specific to cancer, the benefits of sharing will increase for datasets that include more samples. To further explore the benefits of sharing by the shredded compilation route, future experiments should perform the use cases of this section with the output of `BuildOccur` in place of the `Occurrences` data source.

8.8 Future Aims for Biomedical Analysis Support

As noted in Section 8.1, existing biological analysis platforms often provide workflow description languages or domain-specific analysis APIs. Workflow description languages support pipeline construction involving independent tools where each individual component requires manual, tool-specific optimizations. Analysis APIs focus on a specific analytical task and are embedded in host languages like Python. Neither of these solutions provide high-level languages that can both optimize user queries and target distributed processing environments, which is what TraNCE does well. There are many improvements that could be made in TraNCE to better support biomedical analyses. For example, the results here are specific to multi-modal integration of processed genomics datasets, ie. *tertiary analysis*. Upstream pre-processing steps of genomics data could be explored, such as the identification of high-confidence variants; though, there are already many high-quality solutions available for such tasks. This could be supported by further consideration in the interfacing with external tools. The next chapter touches upon this with user-defined functions, but additional compute environments, tool repositories, and data access mechanisms could be considered.

The burden-based use case is specific to data integration tasks that support downstream learning. This is an initial exploration at providing support for the machine learning life cycle. Current learning applications avoid large-scale matrix construction, and instead use sampling and cross-validation techniques. TraNCE can provide scalable data integration to support inference on a full, aggregated matrix. A system that can support a high-level language, large-scale data integration, and downstream learning tasks provides more opportunities for optimizing the whole inference pipeline. Chapter 9 presents a biomedical case study that considers such optimization opportunities, specific to pushing feature selection filters upstream into a large-scale processing environment.

Chapter 9

Feature Selection Filters: A Biomedical Case Study

This chapter presents a case study for UDFs in the shredded compilation route, focusing on optimizing programs that perform feature selection. Feature selection is a preprocessing technique that selects important features for a specific problem, such as biomedical classification. Given a set of features, their numeric values, and predictor labels for a classification task, the goal of feature selection is to identify the most relevant features for distinguishing between classes. When the irrelevant features are removed, inference is more generalizable and the likelihood of *overfitting* is reduced. Overfitting occurs when a model learns noise in a training set that hinders performance for new datasets. Feature selection is thus an important quality assurance step for supervised classification tasks.

Biomedical classification tasks often use biological entities as features; for instance, gene-based features could be in the order of tens-of-thousands. As discussed in Chapter 8, biological systems are complex and feature sets derived from omics datasets lie in high-dimensional space [124]. This *curse of dimensionality* presents a challenge for feature selection methods applied to biomedical classification tasks.

Feature selection methods can be generally classified as either *filter* or *wrapper* methods. Filter methods use a statistical metric to reduce the score of irrelevant features and return only top performers. Wrapper methods consider sets of features and for

each set fit a supervised model. The term "wrapper" relates to the method providing a feature-selection specific interface to an underlying model, such as a random forest or a decision tree. The subsets are then evaluated by a performance metric depending on the task, such as classification accuracy, calculated from the resulting model. The filter methods often do not capture the relationship between features, but are quick estimates that run fast on large datasets. Wrapper methods rely on expensive search algorithms and are not practical for large dataset. Due to the underlying supervised model, however, these methods can better capture the complexity of the feature space in comparison to filter methods. These trade-offs have inspired the development of feature selection methods that combine both approaches for biomedical application [125, 126].

While wrapper-style feature selection methods are better able to capture complex relationships in comparison to filter methods, the runtime of wrapper-style methods, such as RFE, is dramatically increased in such big biomedical feature spaces. The use of feature selection methods in high-dimensional feature spaces presents an interesting opportunity to explore how pushing filters upstream can provide both runtime and model performance. This chapter revisits the burden-based classification analyses with a focus on feature selection, the related performance impacts, and how this process can be automated and optimized with the `udf` NRC_{agg} construct.

Thesis Motivation. The feature selection case study is designed to investigate the whole of the machine learning pipeline, from data integration to evaluating model accuracy. The data integration tasks focus on both processing nested collections and returning intermediate nested collections that are used in downstream UDFs specific to training. The feature matrix is represented as a one-level nested object with the top-level attributes denoting feature names and the sample identifiers, predictors, and feature values on the lower-level. This is the most efficient way to represent a matrix in TraNCE. This case study focuses on the use of shredding techniques applied to this shallow-nested, matrix representation as well as the use of user-defined functions when data is nested. Here, the UDFs are optimized via hints that signal pushing part of the feature selection calculation into the NRC_{agg} program to filter the feature matrix. The pushing of a partial calculation for the purposes of filtering is referred to here as a *partial filter*. Section 9.2 discusses the partial filters, which are based on mutual information and chi-square feature selection methods. The implementation of the

filters in the framework, including the hint optimizations and advantages of using the shredded compilation route are provided in Section 9.3. Section 9.5 presents a set of micro-experiments designed to evaluate the impacts of partial filters with respect to both runtime and model performance.

The results of the feature selection case study can be summarized as follows:

- The pushing of filters from UDFs upstream in the program adds little overhead to the overall runtime, but further evaluation is required for larger datasets and cluster sizes.
- Partial filter pushing provides an 11× speed-up for filter-based feature selection methods, and enables wrapper-based methods to run that were previously unable to perform at all.
- Feature selection filter pushing from UDFs benefits binary classification methods for single and multi-omics approaches, reporting up to 98.7% accuracy in predicting prostate cancer severity.
- Partial filters pushed from UDFs can increase model performance for multi-classification problems, but more partial calculations should be explored that can better capture the complexity of multi-class prediction.

9.1 Background

This work explores a few feature selection methods from each category. Given their short runtimes, three filter method representatives were chosen based on their similarity to each other [127]: Chi-Square, Analysis of Variance (ANOVA), and Mutual Information (MI). Recursive Feature Elimination (RFE) is the representative wrapper method. MultiSURF [126] is used as a representative integrative approach, specifically designed for biological feature spaces. Each of the feature selection methods are summarized next.

9.1.1 Chi-square

Chi-square feature selection is a filter-based method that uses the chi-square (χ^2) test to measure independence between the feature category and predictor label. The χ^2

statistic is calculated based on expected and observed values; the calculation is now described for a binary class. First, the dot product between the feature values and class is taken. The feature values are summed with respect to each class to get the observed frequencies. The dot product of the observed frequency for class and feature values are taken to produce the expected matrix. The χ^2 statistic is then:

$$\chi^2 = \frac{1}{s}\sum_{k=1}^{n}\frac{(O_k - E_k)^2}{E_k}$$

where s is the number of samples, n is the number of features and O_k and E_k are the observed and expected matrices, respectively. The higher the χ^2 statistic, the higher the dependence between the feature and the response. A p-value test for statistical significance is done with the χ^2 distribution test using 1 degree of freedom for a binary class. Small p-values reject the null hypothesis that the feature and response relationship, as determined by the χ^2, is independent. The chi-square method is then qualifying the relationship between the feature and response based on the class distribution.

9.1.2 ANOVA

Analysis Of Variance is a filter feature selection method that uses the F statistic to measure if a class variable is explained by a feature. A feature with a higher F means that it is a better class discriminator than the features with lower scores. The calculation starts by taking the mean feature values across all classes ($\bar{x}$) and the mean feature values for each class ($\bar{x}_i$). The F statistic is then calculated as:

$$F = \frac{\sum_{i=1}^{l} n^i(\bar{x}_i - \bar{x})^2}{\sum_{i=1}^{l}\sum_{j=1}^{n^i}(x_i - \bar{x}_i)^2/n - 1}$$

where l is the number of classes, n is the number of features, and x is the observed feature value. The ANOVA method is then qualifying the feature-response relationship based on sample mean.

9.1.3 Mutual Information

Mutual Information is a filter feature selection method that measures the dependence between two variables. The calculation of MI is the difference between the measure of

uncertainty and the conditional uncertainty, which is high when values occur at the same probability and low otherwise. For two variables x and y and probability function p the uncertainty measurement is:

$$H(y) = -\sum_{y} p(y)\ log_2\ p(y)$$

and the conditional entropy is:

$$H(y|x) = \sum_{x,y} p(x, y)\ log(\frac{1}{p(y|x)})$$

which gives $MI = H(y) - H(y|x)$. MI is equal to zero if and only if the two random variables are independent, with higher values implying a higher dependency. Features that maximize mutual information are passed for training. The mutual information method is then qualifying features and response based on correlation.

9.1.4 Recursive Feature Elimination

Recursive Feature Elimination (RFE) is a wrapper-style feature selection method that uses an external estimator to weight features, such as random forest. RFE recursively considers smaller and smaller subsets of features over several iterations, technically using a filter-based technique internally. First, the estimator is trained on the initial set of features and the importance of each feature is obtained. The importance of each feature is closely related to concepts of collinearity and dependencies that might exist in the model. The "least important" features are then removed from the current set of features. This procedure is repeated on the smaller set with the less important features removed until the desired number of features to select is eventually reached. The recursive feature elimination method is thus qualifying features and response based on an underlying statistical model.

9.1.5 MultiSURF

The MultiSURF feature selection method [126] is a Relief-Based Algorithm (RBA), which is a class of filter methods that use a nearest-neighbor approach to capture feature interactions [128] and gene-gene interactions in particular. RBAs differ from other

filter methods in that redundant and uncorrelated features are not eliminated; rather, features are weighted based on their relevance to the response as defined by the relief algorithm. At each relief cycle, the distance between the target and all other instances is calculated. Two nearest neighbors are identified: one with the same class (hit) and one with the opposite class (miss). The weight is increased if the miss differs from the target feature since this suggests a feature is informative; otherwise, the weight is decreased if the hit differs and unchanged otherwise. Many variations of RBAs have been proposed that manipulate the concept of neighbors, the distances between them, and their weights [128].

The MultiSURF RBA uses a distance threshold for assigning neighbors and only near neighbors contribute to weights; the elimination of far neighbor scoring sets MultiSURF apart from the other RBAs [126]. MultiSURF is argued to be the best RBA in terms of both efficiency and detecting interactions.

9.1.6 Feature Selection in the Literature

Feature selection methods have been explored in the literature most often in terms of runtime and model accuracy [127], collectively referred to here as performance. Database-style optimizations applied to feature selection tasks have been explored in the context of machine learning systems. The DFS language, for example, is a declarative approach to feature selection [129]. The optimizer chooses the optimal feature selection strategy for a given workload and a precomputed feature matrix. The COLUMBUS system exploits materialization opportunities for reuse in feature selection workflows [130]. Neither of these systems explore optimizations in the context of the whole pipeline, from feature matrix construction to model performance. To the best of our knowledge, selection pushing has not been explored in the context for feature selection workflows.

9.2 Partial Filter Calculations

This section describes the partial filter calculations that are based on feature selection methods, referred to as *feature selection filters*. Two filters are presented in this

section - chi-square and correlation. The correlation method is based on the mutual information feature selection method. Correlation is an example of how a full calculation, i.e. calculating the Pearson correlation coefficient precisely without estimations, can be pushed into a query plan. Chi-square is provided as an example of how partial calculations can also be pushed. The general idea is that light-weight calculations can be pushed into the query plans in order to remove features that have a high-likelihood of being thrown out from a downstream feature selection method. The term *pushing* is used loosely; the filters discussed here are partial filters in that they are not performing the full filtering that occurs in feature selection. The use of partial filters provides flexibility to use in combination with other feature selection methods. For instance, these filters will speed up wrapper-style methods and also localize filtering and embedding methods to increase the quality of their feature sets. Further performance advantages provided by the upstream distributed environment are discussed in Section 9.3. The correlation filter is discussed next, followed by the chi-square filter.

9.2.1 Correlation Filter

The correlation filter is based on the mutual information feature selection method discussed in Section 9.1.3. For a predictive model, the goal is to reduce the feature space and end up with features that maximize the mutual information between the selected features - a subset of the initial features - and the target variable. Due to the exponential number of possible feature combinations, maximizing this quantity is an NP-hard optimization problem. The correlation calculation was thus chosen as an approximation of this filter; this is a light-weight calculation that can be pushed into a query plan without the complexity overhead of mutual information calculation.

The correlation coefficient, ρ is defined as:

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

The above calculation shows how ρ is calculated between a feature X, and the target variable Y. $\bar{x}$ is the mean value of the feature calculated from the sum of values across all samples divided by the total number of values, n. For $0 < i <= n$, x_i is a feature

value for a given sample and y_i is the label value for the same sample at i. The above calculation is then repeated for all features eventually returning a list of correlations between all features and the target variable.

The assumption is that the most important features will be the ones with a relatively large correlation, be it positive or negative. Correlation values fall somewhere in between, -1 and 1, $-1 \leq \rho \leq +1$, and so we want to keep the features that have values close to the two extremes, -1 and +1. The correlation filter selects features based on the default threshold $|\rho| \geq 0.1$.

The correlation calculation is an ideal candidate because the full calculation can be pushed without any heuristics. This is the best case situation for filter calculations. For some calculations, only a partial calculation can be pushed to perform the filtering, such as the chi-square filter discussed next.

9.2.2 Chi-square Filter

The chi-square filter is a partial calculation of the chi-square feature selection method discussed in Section 9.1.1. Due to the presence of hypothesis testing and the use of p-values, it was not possible to push the exact chi-square filter calculation upstream. The chi-square filter thus only involves the calculation of a variation of χ^2 value, referred to as $\tilde{\chi}^2$ and defined as:

$$\tilde{\chi}^2 = \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k}$$

This is the partial calculation that will be used within the pushed filter.

In the above equation, $\tilde{\chi}^2$ is calculated for all features, with n observations for each of the features. O_k denotes the observed value of the feature and E_k represents the expected value. To give a simple example of how this is calculated, consider Table 9.2.2 below for a *binary* target variable, Y, with n unique values.

Observation	**Y**	$Feature_1$	**...**	$Feature_n$
1	0	4.6	...	1.0
2	1	4.2	...	1.4
3	1	4.5	...	1.6
4	1	5.0	...	2.0
5	0	5.3	...	2.5
6	0	4.0	...	0.9

Table 9.1: $\tilde{\chi}^2$ **calculation example**

For $Feature_1$, we first perform a dot product between the feature and target. For Y = 0, we have a sum of 13.9 and for Y = 1 a sum of 13.7. Next, the class frequency is calculated: P(Y=0) = 0.5 and P(Y=1) = 0.5 since half of the observations have a 0 label. The sum of all feature values is then calculated: count = 13.9 + 13.7 = 27.6. The dot product is taken to calculate the expected matrices. For Y = 0, the observed value of $Feature_1$ is P(Y=0)*count = 0.5 * 27.6 = 13.8, and for Y=1, P(Y=1)*count = 0.5 * 27.6 = 13.8. Finally, the $\tilde{\chi}^2$ value is calculated: $\tilde{\chi}^2 = \frac{(13.9-13.8)^2}{13.8} + \frac{(13.7-13.8)^2}{13.8}$ = 0.001449. The same procedure is then repeated for all features.

For the original chi-square feature selection, this value is used for hypothesis testing. A high $\tilde{\chi}^2$ value indicates that the hypothesis of independence is incorrect, and hence the higher the $\tilde{\chi}^2$ value the more the feature is dependent on the response variable, and so can be selected for model training. A default threshold for the chi-square value is set to ≥ 0.1, where features satisfying this threshold survive the filter.

The experimental results in Section 9.5 present the impacts on performance for various values of these thresholds for both the correlation and chi-square filter. The next section discusses how these filters are implemented within the TraNCE framework.

9.3 Implementation

As mentioned in the beginning of this section, case studies of this chapter focus on NRC_{agg} programs that produce feature matrices for a downstream UDF that performs inference with feature selection. The construction of feature vectors is then performed

within a distributed environment, whereas the UDF executes in a local Python environment. The feature selection filters are implemented in Spark/Scala, since they are designed to operate in the distributed environment where the NRC_{agg} is executed prior to any externally applied UDF. These partial filters reduce the data before it is passed to an external library where all optimization opportunities are lost. Thus, this process partially pushes a local, non-distributed feature selection method upstream into the distributed evaluation of the NRC_{agg} program.

In addition to the benefits of the shredded variant discussed in previous chapters, the feature selection filters leverage three aspects of the shredded compilation route.

i External statistical libraries often require flat inputs. If the data is nested, then the user is responsible for transforming the nested object into a flat representation. A UDF can use the shredded representation directly, applying the operation only to a specific level without any pre-processing.

ii When certain levels of the shredded representation are not used at all, then their materialization will be avoided due to the lazy evaluation of Scala.

iii The label-partitioning guarantee can be advantageous for distributing the partial filter calculations. For example, the filters used in this case study assume the data is distributed by feature, which is guaranteed by compiling the input program with the shredded compilation route.

The filter implementations are now presented by example. The example describes the general structure of the use cases explored in the case study and highlights the shredding-specific aspects from the above list.

Example 8. Consider the burden programs from Section 8.5, which produce top-level feature identifiers with nested feature values for each sample. The shredded `GeneBurden` program that uses the `Occurrences` data source, including the downstream UDF is given along with the corresponding shredded function definition in Figure 9.1. The `first` input parameter, ie. $\textsc{MatDict}_{\texttt{burdens}}$, in the shredded function definition is the only shredded input that is used in the downstream UDF. This is passed directly into the Python context switching function without any additional transformations (i). When this is executed, the lazy evaluation of Scala will not materialize the input associated to the `top` parameter. The shredded compilation will thus

```
TopBag ⇐
for g^F in MatGenes union
  {⟨gene := g^F.gene, burdens := Label(g^F.gene)⟩}

MatDict_burdens ⇐
  sumBy^score_{lbl,sample,toutcome}(
    for o^F in MatOccurrences union
      for t^F in MatLookup(MatOccurrences_candidates, o^F.mutations) union
        for s^F in Samples union
          if s^F.sample := o^F.sample then
          {⟨lbl := Label(t^F.gene), sample := o^F.sample, toutcome := s.^F toutcome,
               burden := t^F.impact⟩})⟩}

Accuracy ⇐ udf^trainudf_{sample,toutcome,lbl,score,MI,corr}(TopBag, MatDict_burdens)
```

(a) Shredded `GeneBurden` program with call to `trainudf`.

```
def trainudf(features, row, lbl, feature, value, filter):
    # load necessary imports
    from keras.models import Sequential
    import ...

    # access inputs registered as temporary views
    data = sqlContext.table(features)
    df = data.toPandas()
    ...

    # pivot dataframe
    df = df.pivot(index = [row, lbl], columns = feature, values = value).fillna(0.0)
    ...

    # feature selection
    if feature_method == "ANOVA":
      topk_features = topKFeaturesANOVA(X, df, ifPlot = False, topK = 200)
      ...

    # build train set
    X_train, y_train = ...

    # fit model, get accuracy
    history = model.fit(X_train, y_train, validation_split = 0.30, epochs= 10)
    acc.append(history.history['accuracy'])
    ...
    return accuracy
```

(b) Shredded function definition of `trainudf`.

Figure 9.1: Example shredded NRC_{agg} program and shredded function definition for `trainudf`.

avoid materializing a shredded input (ii). In the shredded program, the lower-level $\textsc{MatDict}_{\texttt{burdens}}$ has a label-based partitioning guarantee that ensures that the materialized result will be partitioned by each `gene` feature (iii), ie. $\texttt{lbl} := \texttt{Label}(t^{F}.gene)$. The distribution by feature is optimal for distributing the partial calculations, such that the filter can work in parallel across all partitions without the need to shuffle data.

The desired feature selection filter calculation and the thresholds are applied at each partition. The use of a threshold rather than a top-value approach - commonly used in feature selection methods - avoids sorting globally across all partitions.

The surviving features are collected into a local set and broadcasted to each partition, applied as a local filter to each partition of the first-level dictionary. The implementation of the filter calculation is required since partitions are immutable: filtering cannot be directly applied on the same partition as the original feature matrix. Finally, the feature vectors surviving the filter are passed to the UDF, the top features are chosen from the feature selection method, training is performed and the accuracy is returned.

Note the additional parameter `corr` within the `trainudf` function call. This parameter is provided by the user and serves as an optimization hint. The implementation of the the feature selection filter hints are discussed next.

9.3.1 Hints

The feature selection filters are applied when a hint is referenced as a parameter of a UDF: `corr` and `chisq` for the correlation filter and the chi-square filter, respectively. These filter hints can be used with any downstream analysis that uses a feature selection method. Threshold can be provided as an additional parameter, using defaults otherwise. If no hint parameter is provided, then no filter will be applied. The hint is passed through with the parameters of a UDF, following UDF compilation discussed in Section 6.3.1 and Section 6.3.3. The optimizer module checks each UDF node in the query plan for a hint parameter and injects the filter operator associated to the hint into the query plan as described in Section 6.3.4. The implementation of the correlation and chi-square filters can be found in the code repository [30].

An additional feature filter is provided that removes feature values below a certain threshold, referred to as the value filter (FVALUE). This is based on the assumption that feature selection filters focus on the strength of the dependency between a feature and a target variable. The filter is applied after the initial feature aggregation and before the filter is applied, and is particularly useful when the feature matrix is large or cluster resources are low. When used in combination with the feature selection filters, the filter is denoted as CHISQ+ and CORR+. Overall, these additional filters are an initial exploration into pushing parts of the feature selection filters further upstream into the query plan. The programs of the case study that leverage these filters are presented next.

9.4 Use Cases

This section presents the programs of the feature selection filter case study. The programs extend upon the gene burden analyses from Section 8.5, integrating biomedical data sources to construct feature matrices for use in downstream classification UDFs. The first three programs, presented in Section 9.4.1 perform binary classification to predict severity in prostate cancer using Gleason score as the predictor label. The gene burden matrices are constructed using somatic mutation impact, gene expression, and a combination of the two for the multi-omics case. The final program in Section 9.4.2 is based on the tumor site of origin prediction from Section 8.5, using somatic mutation impact values rather than raw mutation counts. The programs are presented in Figure 9.2, using parameters `fsel` for feature selection and `hint` as placeholders; for example, `fsel` = "ANOVA" and `hint` = "chisq" represents a call to `trainudf` with the ANOVA feature selection method and the chi-square partial filter pushed.

9.4.1 Binary Classification: Prostate Severity

The prostate severity prediction analyses are used as a representative binary classification problem. Gleason score is a common grading system for prostate cancer [131] that assigns a severity measure to a sample. There are several types of grading systems that summarize a cancer as being highly aggressive or less so. For this analysis 0 is a low-grade prediction and 1 is a high-grade prediction. This is identified by the

```
GeneImpactBurden ⇐
 for g in Genes union
  {⟨gene := g.gene burdens := sumBy^{score}_{sample,gleason}(
   for s in Samples union if s.tumorsite == "Prostate" then
     for o in Occurrences union if o.sample == s.sample then
       for t in o.candidates union if g.gene == t.gene then
        {⟨sample := s.sample, gleason := s.gleason, score :=
          if t.impact == "HIGH" then 0.8
          else if t.impact == "MODERATE" then 0.5
          else if t.impact == "LOW" then 0.3 else 0.01⟩})⟩}

Accuracy ⇐ udf^{trainudf}_{sample,gleason,gene,score,fsel,hint}(GeneImpactBurden)
```

(a) Somatic mutation impact burden based on impact values in `Occurrences`.

```
GeneExprBurden ⇐
 for g in Genes union
  {⟨gene := g.gene burdens := sumBy^{score}_{sample,gleason}(
    for s in Samples union if s.tumorsite == "Prostate" then
      for e in GeneExpr union if e.aliquot == s.aliquot then
       {⟨sample := s.sample, gleason := s.gleason, score := e.fpkm⟩})⟩}

Accuracy ⇐ udf^{trainudf}_{sample,gleason,gene,score,fsel,hint}(GeneExprBurden)
```

(b) Gene expression burden based on FPKM values of `GeneExpr`.

```
GeneMultiBurden ⇐
 for g in Genes union
  {⟨gene := g.gene burdens := sumBy^{score}_{sample,gleason}(
   for s in Samples union if s.tumorsite == "Prostate" then
     for o in Occurrences union if o.sample == s.sample then
       for t in o.candidates union if g.gene == t.gene then
         for e in GeneExpr union if e.gene == g.gene && e.aliquot == s.aliquot then
          {⟨sample := s.sample, gleason := s.gleason, score :=
            if t.impact == "HIGH" then 0.8
            else if t.impact == "MODERATE" then 0.5
            else if t.impact == "LOW" then 0.3 else 0.01⟩})⟩}

Accuracy ⇐ udf^{trainudf}_{sample,gleason,gene,score,fsel,hint}(GeneMultiBurden)
```

(c) Multi-omics burden: mutation impact combined with gene expression.

```
GeneBurdenPancan ⇐
 for g in Genes union
  {⟨gene := g.gene burdens := sumBy^{score}_{sample,gleason}(
   for s in Samples union
     for o in Occurrences union if o.sample == s.sample then
       for t in o.candidates union if g.gene == t.gene then
        {⟨sample := s.sample, tumorsite := s.tumorsite, score :=
          if t.impact == "HIGH" then 0.8
          else if t.impact == "MODERATE" then 0.5
          else if t.impact == "LOW" then 0.3 else 0.01⟩})⟩}

Accuracy ⇐ udf^{multiudf}_{sample,tumorsite,gene,score,fsel,hint}(GeneBurdenPancan)
```

(d) Multi-class prediction of tumor site based on somatic mutation burden.

Figure 9.2: The NRC_{agg} programs for the feature selection case study.

`gleason` attribute from the `Samples` data source. Various single genomics datasets have been used to predict Gleason score, a method that has become more common over the past few years [132]. Collectively, these programs evaluate two single omics datasets, used independently as well as their combination.

Single-Omics: Gene Expression. Gene expression is a common genomics data set to use in the analysis of prostate cancer given the known prostate-specific biomarkers [131]. The `GeneExprBurden` program (Figure 9.2b) is based on such analyses, which uses the FPKM values from the `GeneExpr` dataset as feature values. The features are identified by top-level gene names and a nested collection of FPKM feature values for each sample and associated predictor label (`gleason`), following the shallowly-nested matrix representation described at the beginning of this chapter. The materialized matrix is passed to the `trainudf` function to perform classification with the feature selection method identified by `fsel` and the optional partial filter hint `hint`. The output of the program is the average testing accuracy after 100 epochs of cross-validation training.

Single-Omics: Somatic Mutation Impact. The `GeneImpactBurden` program (Figure 9.2a) builds a gene burden feature matrix from the somatic mutation impact values in `Occurrences` filtered for prostate samples only. To the best of our knowledge, gene burden with somatic impact values has not been used in the prediction of prostate cancer severity. As in the `GeneExprBurden` program, the feature matrix has for each top-level gene feature, an aggregated mutation impact value for each sample and associated predictor label. The `GeneImpactBurden` matrix is passed into the `trainudf` training UDF, which performs classification and returns the testing accuracy.

Multi-Omics: Combined Expression Impact. The multi-omics program `GeneMultiBurden` (Figure 9.2c) combines somatic mutations and gene expression for the prostate severity prediction task. Each mutation impact value is multiplied by the corresponding `fpkm` value and this is summed across each gene for each sample. The resulting matrix has the same shallowly-nested form as the single-omics programs - top-level gene feature names and a nested collection of multi-omics feature values for each sample with the associated predictor label. The matrix is passed to `trainudf`, classification is performed, and the testing accuracy is returned.

9.4.2 Multi-class Prediction: Tumor Site of Origin

The multi-class prediction program (Figure 9.2d) is designed to perform tumor site prediction in a pancancer cohort. These extend the burden-based analyses of Section 8.5, improving the feature selection and downstream training methods as well as using mutational impact values instead of raw counts. The `GeneBurdenPancan` program builds a pancancer feature matrix in the same fashion as the programs from the binary classification analysis. The matrix has top-level gene feature names, and a nested collection of burden-based impact feature values for each sample along with the associated tumor site of origin predictor label (`tumorsite`). The resulting matrix is passed to the `multiudf` which is similar to `trainudf`, but for multi-class prediction. There is also a `1vrestudf` that will perform one-vs-rest analysis on the `GeneBurdenPancan` matrix. Each of these UDFs perform the relevant classification and return the model accuracy.

9.5 Experiments

This section contains a set of micro-experiments that explore the performance of feature selection filters in user-defined functions that perform classification. Binary and multi-class predictions are evaluated using the single and multi-omics gene burden programs from Section 9.4. Both runtime and model accuracy is used to measure performance. The run without the feature selection filter is used as a baseline. The experimental setup is described in Section 9.5.1. The binary classification results are presented in Section 9.5.2, followed by the multi-class prediction results in Section 9.5.3. Section 9.5.4 presents the gene set enrichment results for the feature sets from the best performing models from the two analyses.

9.5.1 Experimental Setup

The experiments in this section use the programs from Section 9.4 to evaluate the runtime and model performance for the feature selection filters. Throughout the experiments, the program runs are referred to by name denoted with the feature selection method (`fsel`) and partial filter hint (`hint`). The hint parameter is annotated with

the number of features n that survived: $\texttt{GeneImpactBurden}^{\texttt{fsel}}_{\texttt{hint:n}}$. The feature selection methods include chi-square (ChiSq), ANOVA (ANOVA), mutation information (MI), and recursive feature elimination (RFE). All methods were implemented with `sklearn.feature_selection` from Python [62]. RFE uses a random forest estimator with step size 10, denoting the number of features to remove at each iteration. Each feature selection method returns the top 200 out of 55000 features (total number of genes in `Genes`) based on the preliminary exploration in Appendix A.4. The feature selection partial filters used are chi-square (chisq), correlation (corr), value filter (fvalue), chi-square with the value filter (chisq+), and correlation with the value filter (corr+). The program without the filter applied is used as the baseline. For example `GeneImpactBurden` with chi-square feature selection method and correlation filter 10K features surviving is represented as $\texttt{GeneImpactBurden}^{\textsc{ChiSq}}_{\textsc{corr}:10K}$ with $\texttt{GeneImpactBurden}^{\textsc{ChiSq}}$ as the baseline. RFE is the representative wrapper-style feature selection method, all others are filter methods (Section 9.1).

The classification model is a fully connected, feed forward neural network (NN) with three layers. For binary classification, the first two layers have a LeakyReLU activation function with $\alpha = 0.05$ and a dropout layer with the dropout rate being 0.15. The third and final output layer has a sigmoid activation function. The model is trained using the binary cross-entropy loss function and Adam optimizer for the gradient descent optimizer. For multi-class, the first layer has dropout rate 0.30 and the second layer has dropout rate 0.20; both layers have LeakyReLU activation functions with $\alpha = 0.05$. The final output layer has a softmax activation function and categorical cross-entropy loss function and Adam optimizer for the gradient descent optimizer. Each experiment splits the data 70/30 for training and testing, 30% of the training set is used for validation.

All experiments were run using the shredded compilation route on a single node cluster with Spark 3.1.2 / Scala 2.12, one worker, 8 cores, and 100 G memory. The prostate analysis uses 1.2G `Genes`, 167M of `GeneExpr`, 1.2M `Samples`, and 4.7G of `Occurrences`. The pancancer analysis uses the same gene set, 37M `Samples`, and 6.7G `Occurrences`. The pancancer `Occurrences` dataset includes only the mutations processed with a single variant caller, whereas prostate uses all available somatic mutations.

Each experiment reports the runtime as the time to build the feature vectors, per-

form the partial filter (if any), and run the downstream feature selection. Both chi-square (CHISQ) and correlation (CORR) filters were used to assess runtime using 10K and 20K surviving features, ie. CHISQ:10K, CHISQ:20K, CORR:10K, and CORR:20k. The 20K upper-bound was chosen based on the performance of RFE, which failed to complete and crashed for an number of input features above 20K. Model performance is reported in terms of testing accuracy. Any experiment specific details are provided in the relevant sections.

9.5.2 Binary Classification: Prostate Severity

This set of experiments evaluates the runtime and model performance for the prostate severity binary classification analyses, ie. `GeneImpactBurden`, `GeneExprBurden`, and `GeneMultiBurden` programs. The runtime results are presented first, followed by the model performance for each program.

Runtime. This experiment evaluates the runtime performance of the feature selection filters compared to when no filters are pushed (baseline) for binary classification tasks. Runtime is provided as a combination of total time to build the feature matrix, perform the filter, and run the feature selection method. Figure 9.3 presents the runtime results for `GeneImpactBurden`, `GeneExprBurden`, and `GeneMultiBurden` programs. These results include runs with all feature selection methods for no filter and the chi-square and correlation filters each with 10K and 20K surviving filters. The `GeneMultiBurden` program has longer feature matrix construction running times since it is integrating additional information from `Samples` as well as the `GeneExpr` data source.

The ChiSq and ANOVA methods are quick to run, the partial filters adding only a slight overhead in comparison to the baseline. The time to run the filters is proportional to the size of the lower-level dictionary of each matrix. `GeneExprBurden` is the largest with 5.5 million tuples, `GeneImpactBurden` with 220 thousand tuples, and `GeneMultiBurden` with 95 thousand tuples. The 20K filter on `GeneExprBurden` takes around 30 seconds, whereas the 10K filter takes about 15 seconds. The time to run the filter on the smaller dictionaries is less, with the 10K filter running in half the time as the 20K. The decrease in runtime for the number of features returned suggests that the broadcasted feature set, and not the filter calculation itself, is the additional

overhead. Any overhead incurred by the filter is expected to be a payoff that increases model performance, which is confirmed in the next experiment.

The long running feature selection methods highlight the runtime performance advantages for applying partial filters. RFE is only able to run to completion when filters are applied. Both MI and MultiSurf can run to completion without filters, but faster runtimes are observed when the filters are applied. Overall, the results show up to a 11$\times$ speed-up for filter-based feature selection methods and are able to run wrapper methods to completion that were previously unable to perform at all.

Model Performance. The accuracy is evaluated for each binary classification program with and without feature selection filters, which predict the severity of prostate cancer using two single-omics methods (`GeneImpactBurden`, `GeneExprBurden`) and a multi-omics approach that combines the two (`GeneMultiBurden`). Each accuracy is the average over 100 runs and 100 epochs per run. Any reference to the greatest increase in performance does not include the runs that were unable to run without a filter.

Accuracy Report Details:
Table 9.5.2 reports the `GeneImpactBurden` accuracies for chi-square, ANOVA, mutual information and RFE feature selections with and without filters pushed. The chi-square thresholds applied for each number of surviving feature group were $\tilde{\chi}^2 > 0.088$ for 20K, $\tilde{\chi}^2 > 0.28$ for 10K, $\tilde{\chi}^2 > 0.86$ for 5K, $\tilde{\chi}^2 > 2.06$ for 2K and $\tilde{\chi}^2 > 3.56$ for 1K. The correlation filter used low correlation $|r| > 0.10$ for 20K, moderate correlation $|r| > 0.20$ for 10K, and high correlation $|r| > 0.90$ for 5K. The absolute value is used to capture both positive and negative relationships. All remaining features after the high correlation threshold were 1 or -1 so could not be filtered further.

Table 9.5.2 reports the `GeneExprBurden` accuracies for chi-square, ANOVA, mutual information and RFE feature selections with and without filters pushed. The chi-square thresholds applied were $\tilde{\chi}^2 > 280$K for 20K, $\tilde{\chi}^2 > 3.3 \times 10^6$ for 10K, $\tilde{\chi}^2 > 12 \times 10^6$ for 5K, $\tilde{\chi}^2 > 32 \times 10^6$ for 2K and $\tilde{\chi}^2 > 52 \times 10^6$ for 1K. The correlation filter used $|r| > 0.10$ for 20K, $|r| > 0.15$ for 10K, $|r| > 0.20$ for 5K, $|r| > 0.40$ for 2K and $|r| > 0.95$ for 1K. Note the FPKM values used in `GeneExprBurden` returned a wider range of correlation values than `GeneImpactBurden`.

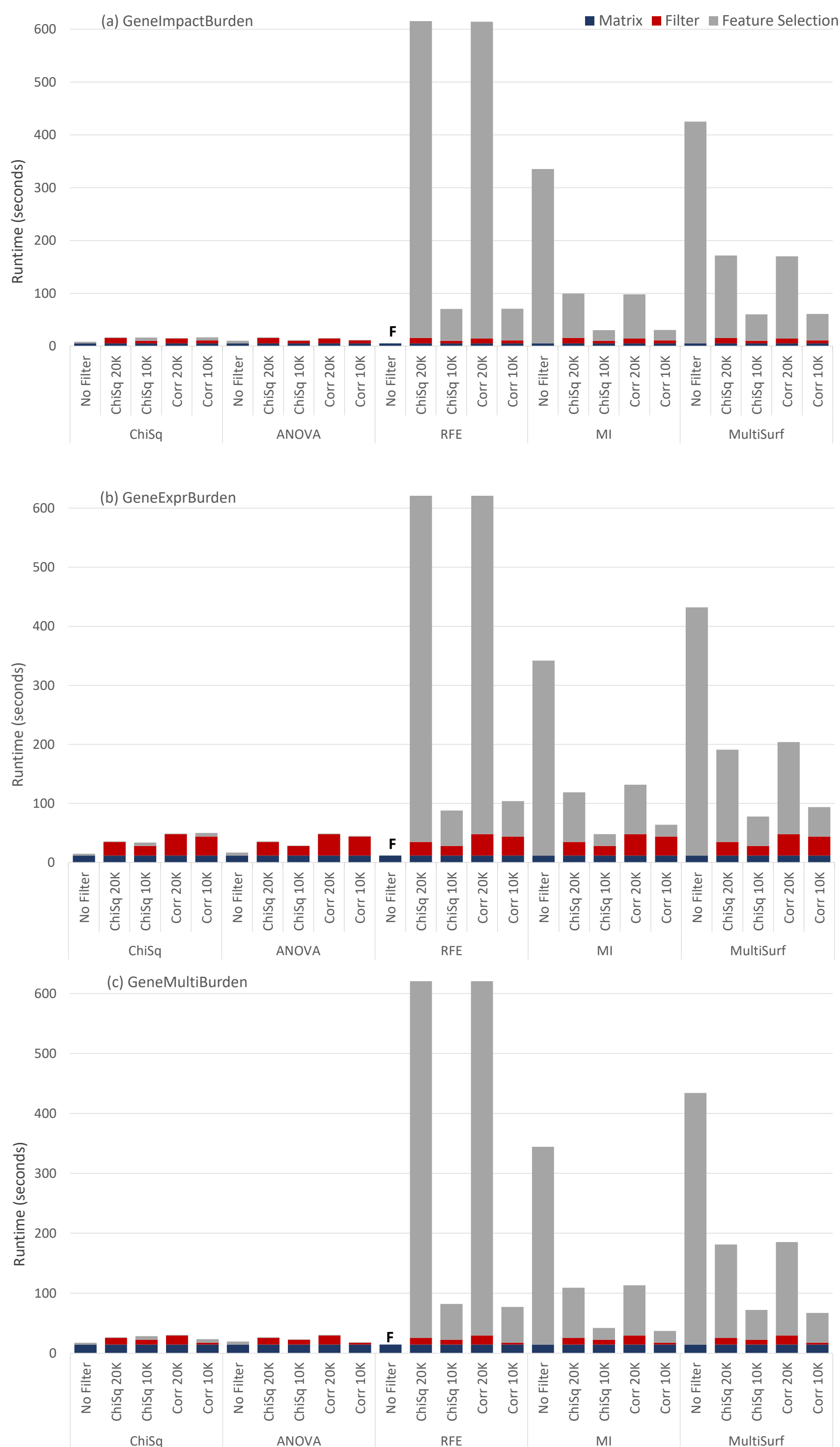


Figure 9.3: Runtimes of the feature vector construction, feature selection methods, and applied filter for `GeneImpactBurden`, `GeneExprBurden`, and `GeneMultiBurden` programs.

Table 9.5.2 reports the MultiSurf based accuracies for the `GeneExprBurden` and `GeneMultiBurden` programs. This method was evaluated due to the past use in biological applications. The results were underwhelming for the burden case studies, but is included for completeness.

Table 9.5.2 reports the `GeneMultiBurden` accuracies chi-square, ANOVA, mutual information and RFE feature selections with and without filters pushed. The chi-square thresholds were $\tilde{\chi}^2 > 0.04$ for 20K, $\tilde{\chi}^2 > 0.15$ for 10K, $\tilde{\chi}^2 > 0.40$ for 5K, $\tilde{\chi}^2 > 1.10$ for 2K, and $\tilde{\chi}^2 > 2.09$ for 1K. The correlation filter used low correlation $|r| > 0.55$ for 20K and high correlation $|r| > 0.90$ for 15K. Similar to `GeneImpactBurden`, all remaining features after the high correlation threshold were 1 or -1 so could not be filtered further. The higher amount of highly correlated features is interesting for the multi-omics case. The integration of the dataset returned almost 5x as many highly correlated features than somatic mutations alone and 15x more highly correlated features than gene expression alone.

Best Performing Models:
For `GeneImpactBurden`, the best performing model was the RFE with 5K features surviving the correlation filter $\texttt{GeneImpactBurden}^{\text{RFE}}_{\text{CORR}:5K}$. This is a 15.7% increase in performance from the best performing model without a filter. For `GeneExprBurden`, the best performing model was the MI feature selection method with 1K features surviving the correlation filter $\texttt{GeneExprBurden}^{\text{MI}}_{\text{CORR}:1K}$. The `GeneMultiBurden` program returned the best performing model overall with the RFE feature selection method with 20K features surviving the correlation filter $\texttt{GeneMultiBurden}^{\text{RFE}}_{\text{CORR}:20K}$.

The best performing model (multi-omics) uses a threshold that is less strict than the best performing single-omics model. This result follows from the observation about multi-omics thresholds in the table details above. The correlation filter is supporting both the runtime and model performance for the multi-omics case; however, as more datasets are added and more correlation is identified then more features will leave the filter. This suggests that improvements to the correlation calculation may be necessary to better capture more complex multi-omics based correlations. In general, the results show that the pushing of filters provides an increase in performance for both single and multi-omics analyses, and highlights the advantages in using multi-omics datasets for prediction.

Chi-Square Filter:
For `GeneImpactBurden` chi-square filter increased nearly all methods, showing higher increases in comparison to the correlation for ChiSq and ANOVA and actually worse performance for some surviving feature groups of MI. This is expected behavior since the chisq filter implements a partial calculation of ChiSq and ANOVA. The chi-square filter performed best for the moderate threshold returning 10K features, after which the partial calculation starts to lose power. Overall, the chi-square calculation provided a 14.9% increase to the chi-square run it originated from, amortizing the slight overhead in runtime. The chisq filter results highlight how the reduction in the input feature space will help the method select higher performing features within the bounds of a moderate threshold.

For `GeneExprBurden` the chi-square filter only returned better accuracy for ChiSq, it actually decreased the performance of all other feature selection methods. The chi-square method without filters had the worst performance of any of the non-filter-based models, so the statistical significance test is likely not a good candidate for extracting high performing features based on gene expression values. On the other hand, the multi-omics `GeneMultiBurden` program has increased performance with the chi-square filter for all methods highlighting the importance of integrating datasets for increased model performance. Further, the chi-square method produced a model with 93.5% accuracy with RFE. This is the highest accuracy returned from the ChiSq filter. Overall, this shows that though the chi-square filter may not be valuable for some single-omics analyses - it is able to show performance improvements in a multi-omics setting.

Correlation Filter:
For `GeneImpactBurden` correlation filter increased the accuracy of all the methods, showing higher increases in comparison to chi-square for RFE and MI methods than ChiSq and ANOVA. This behavior is expected given correlation is an approximation of the MI methodology and the random forest used within RFE is more likely to benefit from correlation measurements than the measure of significance used in chi-square methods. Overall, the correlation filter brought a 16.0% increase to the mutual information method it originated from; the MI run also exhibited the greatest improvement in runtime performance over other methods.

For `GeneExprBurden` the correlation filter increased the accuracy of all the methods, showing higher performance for all the feature selection methods than the chi-square

filter. Though this is counterintuitive to the nature of the calculations, the poor performance of chi-square calculation for gene expression explains this behavior. Following the pattern from `GeneImpactBurden` correlation filter, the `GeneExprBurden` correlation filter improved the mutual information performance by 19.8%. This was the best performing model for gene expression and was based on only 1K feature returned from the filter - the strictest possible threshold. This suggests that it may be possible to push the whole correlation calculation into the feature matrix construction, completely removing the need to run feature selection externally.

For `GeneMultiBurden` the correlation filter with a less strict threshold out performed most of the chi-square methods for all feature methods. The ANOVA method was the only correlation run that performed better with a more strict filter. Following the single-omics results, the correlation filter exhibited the greatest improvement with the mutual information method - other than the best performing model RFE that reported the best performance for all binary classification results. Overall, these results highlight how pushing partial calculations can improve both model performance and runtime, enabling the best performing feature selection methods to run which would have been unable to perform otherwise.

	ChiSq		ANOVA		RFE		MI	
	CHISQ	CORR	CHISQ	CORR	CHISQ	CORR	CHISQ	CORR
No Filter	60.1%		78.3%		-		62.7%	
20K	74.4%	75.2%	83.8%	78.6%	84.8%	90.2%	59.3%	70.7%
10K	74.1%	68.6%	83.9%	78.1%	88.0%	93.1%	66.6%	70.3%
5K	75.0%	62.1%	70.9%	91.2%	79.5%	**94.0%**	52.3%	78.7%
2K	74.9%	-	71.8%	-	58.3%	-	52.9%	-
1K	62.2%	-	77.6%	-	58.3%	-	55.2%	-

Table 9.2: Single-omics mutation impact burden accuracies with and without filter pushed.

	ChiSq		ANOVA		RFE		MI	
	chisq	corr	chisq	corr	chisq	corr	chisq	corr
No Filter	59.8%		80.7%		-		67.8 %	
20K	60.3%	61.1%	76.8%	83.3%	56.7%	78.3%	62.2%	77.5%
10K	61.1%	62.2%	74.5%	83.3%	62.2%	76.3%	61.5%	83.4%
5K	61.3%	70.7%	74.5%	83.4%	62.2%	78.6%	61.5%	84.2%
2K	57.6%	70.6%	70.4%	83.0%	62.6%	87.4%	51.9%	87.2%
1K	58.9%	74.7%	58.8%	83.2%	54.8%	84.0%	54.6%	**87.6%**

Table 9.3: Single-omics gene expression accuracies with and without filter pushed.

	ChiSq		ANOVA		RFE		MI	
	chisq	corr	chisq	corr	chisq	corr	chisq	corr
No Filter	68.6%		80.9%		-		63.0%	
20K	69.8%	76.2%	80.1%	80.8%	93.5%	**97.8%**	64.3%	75.1 %
15K	-	59.9%	-	86%	-	95.3%	-	71.2 %
10K	68.8%	-	84.6%	-	92.8%	-	58.1%	-
5K	68.4%	-	85.4%	-	85.4%	-	58.1%	-
2K	67.5%	-	88.7%	-	80.0%	-	67.2%	-
1K	69.2%	-	84.1%	-	73.7%	-	46.4%	-

Table 9.4: Multi-omics accuracies with and without filters pushed.

	GeneExprBurden	GeneMultiBurden	
	CORR	CHISQ	CORR
No Filter	71.7%	59.4%	
20K	73.6%	53.8%	60.9 %
15K	-	-	53.8%
10K	66.0%	54.0%	
5K	74.8%	58.0%	
2K	77.0%	56.8%	
1K	72.9%	63.3%	

Table 9.5: MultiSurf results for gene expression and multi-omics analyses.

9.5.3 Multi-class Prediction: Tumor Site of Origin

This set of experiments evaluates the runtime and model performance for the tumor site of origin multi-class prediction analysis from program `GeneBurdenPancan` with the correlation filter and value filter applied (corr+). Recall from Section 9.3.1 that the value filter removes features with values below a threshold (0.01). This filter was necessary for the larger pancancer dataset and restricted cluster size used to run the multi-class experiments. The analysis investigates two approaches for predicting tumor site of origin from nine cancer types, extending on the multi-class and one-vs-rest approaches from Section 8.5 to further explore the impacts of feature selection.

Runtime. This experiment evaluates the runtime performance of the feature selection filters compared to when no filters are pushed (baseline) for the multi-class prediction task. Runtime is provided as a combination of total time to build the feature matrix, perform the filter, and run the feature selection method. Figure 9.4 presents the runtime results for the `GeneBurdenPancan` program. These results include runs with all feature selection methods for no filter and the corr+ filter with 10K and 20K surviving filters. The chi-square filter was not used for multi-class since the correlation filter had better performance overall for binary prediction. The runtimes of feature selection for multi-class problems are in general longer running tasks, so there is relatively less

overhead for the pushed filter for this analysis. Overall, the graph exhibits the same runtime patterns discussed in Section 9.5.2.

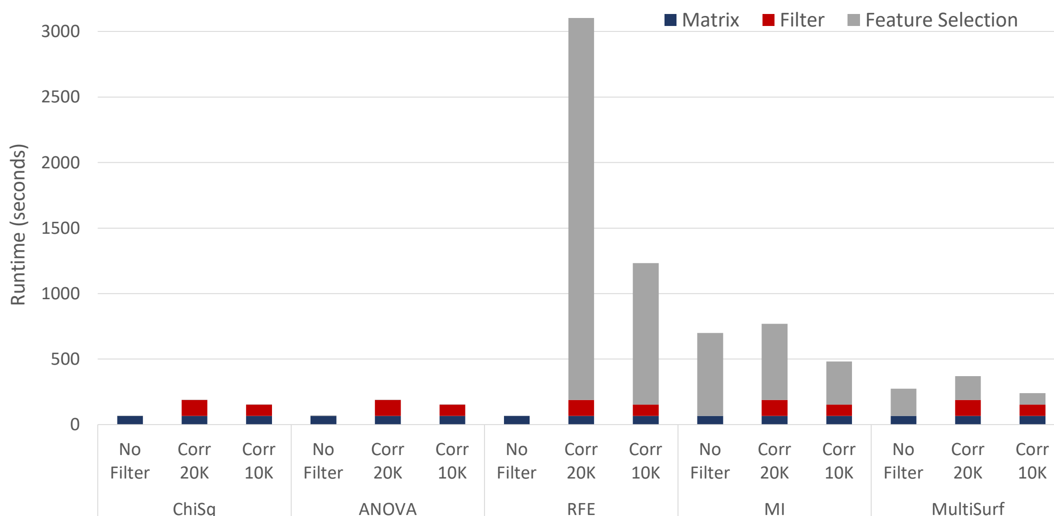


Figure 9.4: Runtimes of the feature vector construction, feature selection methods, and applied filter for `GeneBurdenPancan` program.

Multi-class Performance. The accuracies are evaluated for the multi-class program (`GeneBurdenPancan`) that predicts the tumor site of origin with and without the CORR+ feature selection filter. Each accuracy is the average over 100 runs, 100 epochs per run. Any reference to the greatest increase in performance does not include the runs that were unable to run without a filter.

Table 9.5.3 reports the `GeneBurdenPancan` accuracies for chi-square, ANOVA, mutual information and RFE feature selections with and without filters pushed. The correlation filter used low correlation $|r| > 0.05$ for 20K, moderate correlation $|r| > 0.15 for 15K$, and high correlation $|r| > 0.75$ for 2K. The best performing model was ANOVA without the filter, reporting a higher accuracy than previous reported in Section 8.5. For the multi-class prediction analysis the results show that the already low accuracy declines when the correlation filter is pushed. This is likely related to the complexity of the problem of predicting nine classes, ie. correlation to nine variables is not a good indicator to distinguish between tumor types. Further investigation will be necessary to determine a better calculation to use for multi-class prediction problems.

	ChiSq	ANOVA	RFE	MI
No Filter	45.2%	**49.2%**	-	40.2%
20K	40.2%	45.5%	40.9%	37.2%
15K	36.9%	43.7%	42.9%	37.5 %
2K	32.8%	34.7%	35.9%	33.5%

Table 9.6: `GeneBurdenPancan` results for correlation filter with standard multi-class prediction.

One-vs-Rest Performance. The one-vs-rest approach was first explored in Section 8.5, which exhibited better performance over traditional multi-class approaches in predicting the tumor site of origin. Due to the poor performance observed in the previous multi-class experiment, this set of experiments is designed to take a closer look at pushing feature selection filters in multi-class prediction problems.

The first task is to reduce the complexity of the feature selection calculation to assess the usefulness of pushing a feature selection filter; thus, this experiment pushes only the value filter (fvalue) that removes low-valued features. The accuracies are evaluated using the `GeneBurdenPancan` program with and without the fvalue filter using the ChiSq, ANOVA, and RFE feature selection methods. The setup for the one-vs-rest approach is the same as the standard multi-class, but with a different training approach. Nine binary models are trained one for each cancer type independently. The NN uses a binary cross-entropy loss function and a sigmoid output activation function. More information about how the one-vs-rest approach works can be found in Section 8.5.1.

The merged, overall testing accuracy of the one-vs-rest method may not be fully telling of the quality of a model. This is because high-confidence predictions for a single patient will be washed out by a merged testing accuracy. The prediction of a single patient should be an important factor in assessing the usefulness of a model in clinical practice. Due to this, the performance of one-vs-rest is reported in terms of training accuracy.

The feature selection method is adjusted to fit the nine binary models. Each of the

three feature selection methods outputs an feature set independent of the other classifiers and the binary model is trained on the output of this; thus, each binary classifier has a set of 200 features specific to the classifier being trained. After training each classifier, the classifiers are merged and the performance is evaluated across all samples. The classifiers were trained for 30 epochs, and the training accuracy is reported as an average over 5 runs.

	ChiSq	ANOVA	RFE
No Filter	74.6%	75.7%	-
12K	73.7%	77.0%	**80.3%**
6.5K	65.1%	66.5%	71.7%
3.5K	55.1%	56.2%	62.1%

Table 9.7: `GeneBurdenPancan` training accuracy for value filter with one-vs-rest prediction

Table 9.5.3 presents the training accuracies for `GeneBurdenPancan` with and without fvalue filter pushed. The thresholds used for fvalue 0.01 for 12K, 3.6 for 6.5K, and 7.6 for 3.5K. The value filter did not have much impact on the filter methods chi-square and ANOVA. The performance of the models decrease as the threshold becomes more strict, which is expected since the higher the feature value filter the more likely that a high performing filter will be thrown out. The filter allowed RFE to run, which was previously not able to perform at all, and also reported the best accuracy. The confusion matrix for RFE:12K is provided in Figure 9.5. In the diagonal of the Normalized graph, the accuracies for each cancer type are displayed. False Positive and False Negative rates are displayed in the off-diagonal entries. The final accuracy 80.3% is the sum of the entries in the diagonal over the total number of entries in the Class Counts graph. One-vs-rest localizes the features for each cancer type within each binary model and the value filter reduces the feature space to give more power to the RFE method.

Confusion Matrix - Normalized

TRUE CLASS \ PREDICTED CLASS	0	1	2	3	4	5	6	7	8
8	0.017	0.003	0.009	0.014	0.017	0.032	0.014	0.014	0.880
7	0.005	0.007	0.055	0.012	0.025	0.015	0.010	0.854	0.017
6	0.014	0.005	0.106	0.046	0.023	0.018	0.770	0.018	0.000
5	0.009	0.012	0.045	0.018	0.006	0.886	0.003	0.015	0.006
4	0.029	0.016	0.065	0.078	0.715	0.019	0.013	0.039	0.026
3	0.014	0.028	0.102	0.782	0.019	0.019	0.005	0.017	0.014
2	0.012	0.029	0.851	0.039	0.012	0.028	0.009	0.015	0.005
1	0.006	0.721	0.214	0.019	0.012	0.008	0.008	0.010	0.000
0	0.714	0.013	0.090	0.043	0.013	0.021	0.004	0.051	0.051

Confusion Matrix - Class Counts

TRUE CLASS \ PREDICTED CLASS	0	1	2	3	4	5	6	7	8
8	6	1	3	5	6	11	5	5	307
7	2	3	22	5	10	6	4	344	7
6	3	1	23	10	5	4	167	4	0
5	3	4	15	6	2	295	1	5	2
4	9	5	20	24	221	6	4	12	8
3	6	12	43	330	8	8	2	7	6
2	9	22	638	29	9	21	7	11	4
1	3	347	103	9	6	4	4	5	0
0	167	3	21	10	3	5	1	12	12

Figure 9.5: Confusion matrix of the RFE model. The cancer types are encoded as [Colon = 0, Breast = 1, Lung = 2, Kidney = 3, Stomach = 4, Ovary = 5, Endometrial = 6, Head and Neck = 7, Central nervous system = 8]

Feature Selection in One-vs-Rest:
The choice of using separate feature selection methods for each binary model in the one-vs-rest approach was motivated by preliminary exploration of feature set combinations. Table 9.5.3 reports the accuracy for various feature set combinations using the `GeneMultiBurden`$^{\text{ANOVA}}$ baseline. The top 200 genes were determined independently for each classifier, training each on the individual gene sets (200 individual). Feature selection was then ran for each of the classifiers returning the ANOVA top 20 genes from each and merging before training each classifier (20 per). The classifiers were also trained on the 21 intersecting genes from this combined set (20 per intersect), as well as the non-overlapping (20 per disjunct). The testing accuracy for ANOVA no filter 52.7% with independent feature selection methods performs the best. This is an improvement on the multi-class training accuracy reported above and a 10% increases of the multi-class accuracy reported in Section 8.5.

	200 individual	20 per	20 per intersect	20 per disjunct
ANOVA	**52.7%**	52.0%	41.8%	41.2%

Table 9.8: `GeneBurdenPancan`$^{\text{ANOVA}}$ testing accuracy for gene set combinations.

Binarized Model Performance:
The second experiment looks at a single binarized model for a tumor tissue site in a dataset of multiple cancer types. The advantage of the one-vs-rest approach is that prediction over multiple classes is broken down into independent binary classification problems. This experiment was designed to see if that general idea also applied to the use of feature selection methods.

This experiment builds a binary model for prostate tumor site of origin prediction among six cancer types: stomach, bladder, esophagus, liver, pancreas, and prostate following the `GeneBurdenPancan` program. For this problem, prostate samples have predictor label 1 and the other tissue types have label 0. The model uses the same parameters of the one-vs-rest binary models. Table 9.5.3 reports the accuracy for the model without filters, with value filter only (FVALUE), and chi-square filter with value filter CHISQ+ for ANOVA, RFE, and MI methods. The FVALUE threshold removed all features < 0.02.

The results show that the FVALUE filter enables RFE:12K to run with the highest accuracy, while the accuracy decreases for ANOVA and MI. The application of the CHISQ+ filter brings the accuracy back up for all models, suggesting benefits to using the FVALUE filter with chi-square filter. This filter produces the best performing model RFE:5.5K with 87.8% accuracy. MI does not perform better with either filter likely due to the fact that, unlike correlation filter, the chi-square calculation is not symbiotic with this method. The accuracy drops down again with a more strict chi-square filter. The filters appear to increase the performance of binary tumor site of origin prediction suggesting that feature filters could be of benefit in the one-vs-rest problem. This would require running an individual filter for each tissue site, however, and that will lead to increased runtimes. Further exploration should consider how to optimize filters for binary models in a one-vs-rest approach.

	ANOVA	RFE	MI
No Filter	79.1%	-	80.9%
15K (FVALUE)	72.8%	84.0%	76.1%
5.5K (CHISQ+)	80.2%	**87.8%**	78.7%
2K (CHISQ+)	80.6%	80.2 %	71.2%

Table 9.9: Multi-omics accuracies with and without filters pushed.

9.5.4 Gene Enrichment Analysis

This section discusses the biological relevance of the gene feature sets that returned the best performing models for both the binary prostate cancer severity prediction and the multi-class tumor site of origin prediction (one-vs-rest approach). Gene set enrichment analysis was performed using WebGestalt [133]. Gene set enrichment is a method to measure the correlation of an input gene set to known gene sets. The enrichment analysis discussed here compares against gene sets in the the functional gene ontology database, using the genome reference list and over-representation analysis (ORA). ORA is a statistical method that determines whether genes from pre-defined sets, all human genes in this case, are over-represented in the input gene set [134]. Results are reported based on a False-Discovery Rate (FDR) ≤ 0.05, meaning there is

95% confidence that the results are accurate; this is referred to as a *high-confidence result*. FDR is the expected proportion of false positives among the rejected hypotheses, where the null hypothesis is the gene set being independent of the biological processes and the alternate meaning there is significant correlation. The categorical organization of the gene sets are also provided with respect to biological processes, cellular components, and molecular function.

Prostate Cancer Severity:
The 200 genes from the feature set of the best performing model were analyzed for binary classification ($\texttt{GeneMultiBurden}^{\mathrm{RFE}}_{\mathrm{CORR}:20K}$). The overlap with the second best performing model $\texttt{GeneMultiBurden}^{\mathrm{ANOVA}}_{\mathrm{CORR}:20K}$ was also analyzed, including the 52 shared genes and the 148 genes to RFE only. The gene enrichment analysis for the RFE did not generate any high-confidence scores (all FDR > 0.05); however, the gene set in the overlap identified two high-confidence gene sets both specific to metabolic processes presented in Figure 9.6. Consistent with these results, metabolism was identified as the most represented biological process in the gene ontology classification summary in Figure 9.7. The cellular components and molecular function classifications - membrane, nucleus, and protein binding - are consistent with cancer-specific processes. These results show that multi-omics approaches are able to identify prostate-cancer-specific processes as well as features that are specific to cancer in general.

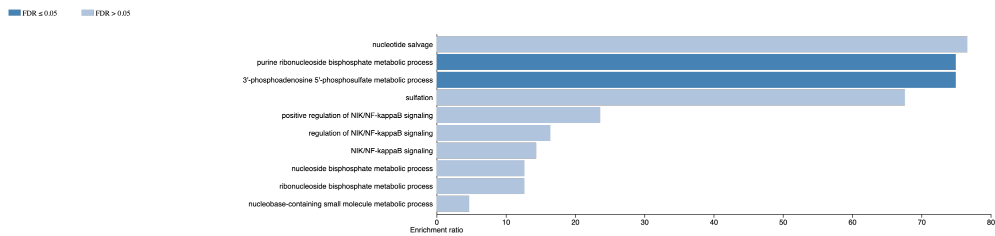


Figure 9.6: Gene set enrichment analysis for overlapping RFE and ANOVA feature sets. High-confidence associations are in dark blue. Generated by WebGestalt.

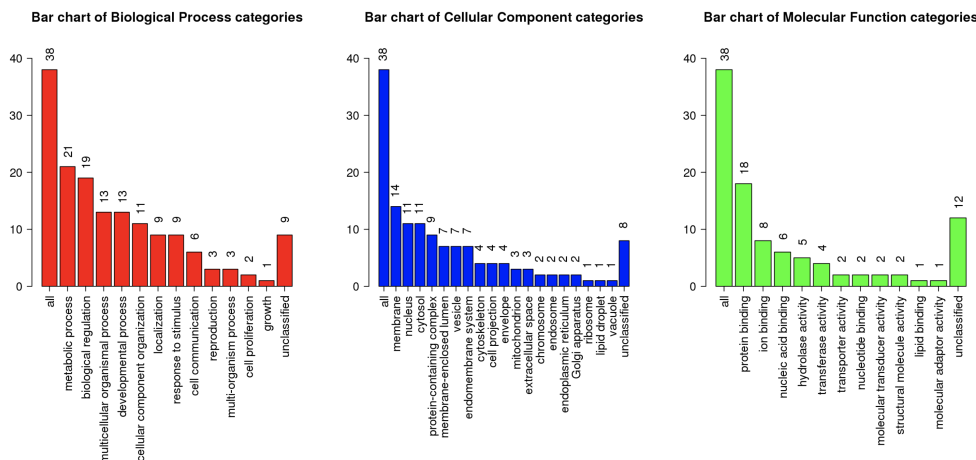


Figure 9.7: Biological processes, cellular components, and molecular function categories for the overlapping gene set of ANOVA/RFE. Generated by WebGestalt.

One-vs-rest Tumor Site Prediction

The overlap between the best performing models $\texttt{GeneBurdenPancan}^{\text{RFE}}_{\text{FVALUE:12}K}$ $\texttt{GeneBurdenPancan}^{\text{ANOVA}}_{\text{FVALUE:12}K}$ was analyzed for the multi-class tumor site prediction problem. There were 114 overlapping genes suggesting that there are many genes that are chosen as high-confidence from both filter and wrapper-based methods. The gene enrichment analysis for the multi-class problem had higher confidence results than the prostate analysis. Ten gene sets had FDR ≤ 0.05, which are noted in the volcano plot in Figure 9.8. The volcano plot shows the negative-log of FDR verses the enrichment ratio, highlighting the degree by which the significant categories stand out (upper corners). The majority of these groups are related to developmental processes and localization. This is confirmed in the gene ontology classification summary in Figure 9.9, denoting biological regulation and developmental processes as highly represented groups. The most represented group for molecular function is protein-binding, inline with cancer and consistent with localization results from the volcano plot.

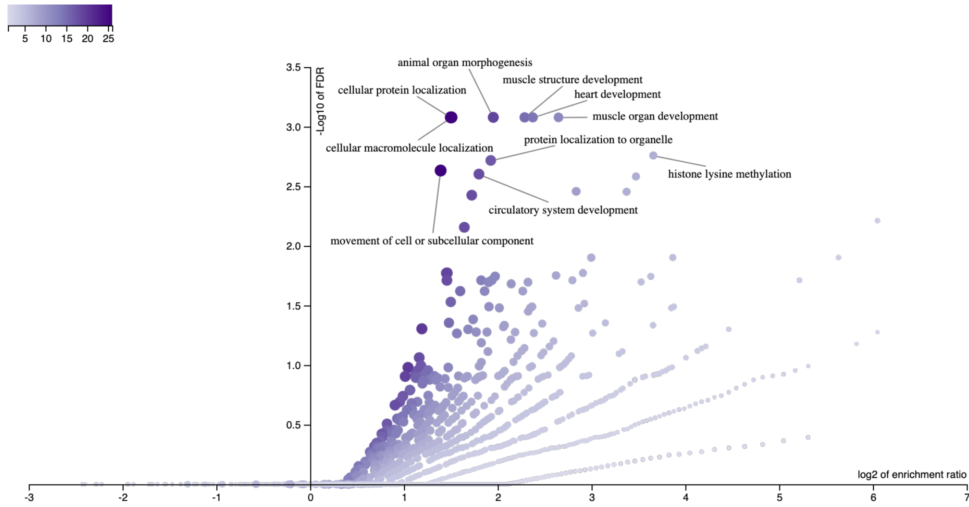


Figure 9.8: Volcano plot for the 114 overlapping ANOVA genes used in the one-vs-rest experiment. High-confidence associations are labeled by definition. Generated by WebGestalt.

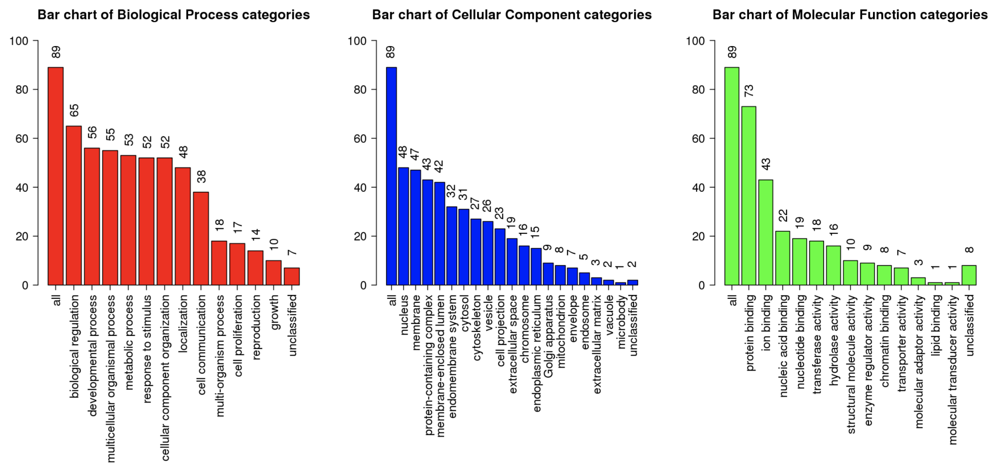


Figure 9.9: Represented biological processes, cellular components, and molecular function categories for the 114 overlapping ANOVA genes used in one-vs-rest. Generated by WebGestalt.

Overall, the results show that while multi-class model performance is lower, the gene sets selected are returned with higher confidence than the binary model. This suggests that the feature selection methods explored are biased to selecting cancer-specific fea-

tures rather than tumor-specific features. Further work should be explored to determine the relationship of this bias to the feature selection methods used and the use of burden-based feature sets.

9.6 Future Exploration

The case study of this chapter has presented how TraNCE can be used to construct and optimize the whole of a machine learning pipeline, from nested data integration to the processing of intermediate nested collections in downstream UDFs. The optimization of UDFs pays specific attention to the use of feature selection filters in the shredded compilation route, and is accompanied with a runtime and model performance evaluation for both binary and multi-class biomedical prediction tasks. While the runtime results show that the feature filters add minimal overhead and increase model performance for binary classification tasks, the results are based on micro-experiments that used a small cluster and reduced datasets. Additional performance gains are anticipated with a larger cluster, but the scalability of pushed filters should be explored further. The filters also enable the use of RFE which ends up performing the best for the majority of the prediction tasks. Most importantly, the best performing model used RFE with the correlation filter pushed as well as a multi-omics feature sets, which is a promising result for both the use of multi-omics datasets as well as feature selection filters. The model performance does have a limitation in that the training and testing is performed on data from the same consortium, and further investigation is required to evaluate this approach for additional prostate datasets.

While the feature selection filters did not perform well for multi-class approaches, there are measurable benefits when using the feature selection filters in the one-vs-rest approach. This suggests that the performance gain for binary prediction is repeatable for other binary prediction tasks, even when the binary models are considered in the context of a multi-class prediction problem; however, the application of the filter to each class-specific model will increase the overall runtime by a factor of the number of classes. The impact of this performance and filter calculations more specific to optimizing multi-class feature selection should be explored.

The biological relevance of the selected features highlighted the bias of selecting

cancer-specific genes. For instance, the one-vs-rest multi-class prediction did not have the best model performance but returned many high-confidence results for gene enrichment, whereas the high-performing prostate severity model did not see many high-confidence results in gene enrichment. There are categorical differences between prostate cancer and pancancer gene sets that should be noted. Most interestingly, the prostate severity enrichment identified metabolic processes, which have been shown to be drivers in prostate-specific and metabolic drugs have been suggested as a treatment option for prostate cancer [135]. This means that genes important to prostate cancer are selected and this is supported by high model performance; however, the gene set enrichment results suggests there is more to be explored on determining the best gene set for prostate severity prediction. Overall, further evaluation should take into account the complexities of the biological system, the relationships between and across cancers, gene-gene relationships, and additional filters that can be pushed for more complicated feature selection tasks.

Chapter 10

Conclusion

This work has presented a compilation framework for processing nested collections on top of distributed processing platforms. While the platform has exhibited promising results, a number of components still remain to explored. Chapter-specific discussions on limitations and future work have been provided throughout the thesis. This final chapter discusses the limitations and opportunities for future work related to language extensions, shredding, optimizations, and data science workloads.

Language features. The source language of TraNCE is a variant of NRC with support for deduplication as well as key-based aggregation and grouping. Deduplication is restricted to be a flat bag to avoid deep-equality. Future work should consider loosening this restriction. Operations that support more advanced statistics should also be considered, such as average, min, and max. Though current support for user-defined functions in the system provides a quick solution to make the language more expressive, the current implementation of UDFs is still very limited. The user is required to supply the shredded UDF definition. An automatic transformation for shredding UDFs should be explored further.

Interfaces to the system. New backends and new frontends are also under investigation. Support for a Flink backend has already been developed and is currently being integrated. While NRC provides a natural syntax for describing transformations over nested collections, it can be argued that other languages have less of a learning curve. This is one reason why many systems support “SQL-like” dialects. The ability to sup-

port additional source syntax can make it easier for both users and other systems to interact with the framework. In addition to the blockly-based web GUI described in Section 6.2, other frontends are being explored, such as NRC translation to SQL and JSONiq.

Richer data models. Some systems that process distributed data, such as DIQL and Rumble, work with a data model that is richer than the nested model. While the results of this thesis bring to light the performance issues of these systems within the bounds of the simpler nested data model, the application of this work to richer models needs to be considered. Support for ordering would require having a list type. Heterogeneous types should also be considered, which could be a challenge for the shredding transformation since label types sanity check lookup operations on dictionaries. The use of shredding with a richer data model should be explored in more depth.

Shredding techniques. The shredded compilation route is implemented as an extension of the standard compilation route, which supports a fair comparison of flattening and shredding techniques from within the same system. As noted in the system evaluations in Chapter 7, the shredded representation shows many benefits. But there are many limitations to consider, especially related to the consumption of shredded representations in other systems. First, current applications that work with nested data expect data in nested form. The results show no additional overhead in unshredding in comparison to other systems; however, there is much to benefit from the shredded representation since returning nested output types is expensive. Second, the current results consider inputs that have already been shredded. The shredding of input data still needs to be explored, including the process behind input shredding and the related overheads. Ideally, data sources and related applications could return shredded representations, which can then be processed using TraNCE and avoid the need to shred inputs altogether. In general, the interaction of shredded representations with external systems, tools, and databases should be explored further.

If applications can produce and consume shredded representations, then it is worth considering how state-of-the-art systems, such as Spark, could benefit from the use of shredding techniques. As we have shown in TraNCE, support for shredding does not need to be all-or-nothing. The shredded representation can also be used in combination with nested data, which is achieved by partial shredding.

$$\mathtt{COP}^{\mathrm{D}} : \langle\, \mathtt{corders}^{\mathtt{fun}} : \mathit{Label} \to \mathit{Bag}(\langle\, \mathtt{odate} : \mathit{string}, \mathtt{oparts} : \mathit{Bag}(\langle \mathtt{pid} : \mathit{int}, \mathtt{qty} : \mathit{real} \rangle),$$
$$\mathtt{corders}^{\mathtt{child}} : \mathit{Bag}(\langle\rangle)\rangle\rangle\rangle)$$

(a) Partially shredded type.

```
let cop^D := COP^D in
⟨corders^fun := λl. match l = Label(cop^F) then
   for co^F in Lookup(cop^D.corders, cop^F.corders) union
     { ⟨ odate := co^F.odate, oparts := sumBy^total_pname(
              for op in co^F.oparts union
                for p^F in Part^F union
                  if p^F.pid == op.pid && o^F.odate == "01 : 03 : 22" then
                    {⟨pname := p^F.pname,
                      total := op.qty * p^F.price⟩}
             )⟩},
corders^child := {⟨⟩ }⟩
```

(b) Partially shredded query.

Figure 10.1: Partial shredding for the COP input and the Totalsrunning example.

Shredding can easily be tuned towards either partially-shredded inputs or partially-shredded outputs. Consider the fully-shredded input of COP that contains a top-level bag, $\mathtt{COP}^{\mathrm{F}}$ and a dictionary tree of $\mathtt{COP}^{\mathrm{D}}$ with two-levels containing $\mathtt{corders}^{\mathtt{fun}}$ and $\mathtt{oparts}^{\mathtt{fun}}$. In practice, the second-level dictionary associated to oparts may contain only a few tuples per label. With this in mind, a user may chose to partially-shred the input to only contain a top-level bag and a one-level dictionary tree. The partially shredded type of COP could leave oparts as a bag type, resulting in the type in Figure 10.1a. This would then require a modification of the materialized output of the query associated to the lower-level dictionary. This modification is exhibited on the Totalsrunning example in Figure 10.1b. What was a previous lookup on $\textsc{Mat}\mathtt{COP}_{\mathtt{corders_oparts}}$ is now just a projection on the lower-level oparts collection. Partial shredding optimizations can avoid shredding small collections that are not the main contributor to the performance impacts discussed in this thesis.

When implementing shredding within the TraNCE framework, aspects of distributed processing influenced the shredding transformation. For instance, the sequential materialization techniques were better designed for distributed processing platforms,

whereas monolithic techniques used in the past were better for relational engines. Monolithic shredding may not always be a loss for distributed processing platforms. Alternatively, the use of domains in the sequential strategy could be highly beneficial for queries with restrictive filters. Further exploration is required to understand the trade-offs of these methods in more detail, in addition to other materialization strategies and specific operations that can make one method perform better than others. For example, the use of domains could be particularly beneficial for queries with restrictive filters, the decision of which could be governed by a cost-based optimization framework.

Optimization. Other than the skew-handling techniques, all framework optimizations are heuristics that are turned on or off based on user input. A crucial issue and a target of ongoing work is the use of cost-based optimization within the framework. To the best of our knowledge, there is no existing cost-based optimization techniques that are specific to nested data processing. The estimator in Spark, for example, does not produce estimates for unnest or nest operations. While future work should consider costing the operations involved in flattening techniques, the lack of exploration in the area of nested collection estimation highlights an added benefit of the shredded representation. Shredding provides a relational representation to a nested schema that can potentially simplify optimizations for nested collection processing down to that of relational optimization techniques. Future work should explore how relational techniques can be applied to the shredded representation.

One open question related to applying relational optimizations to the shredded representation is the impact of value unshredding. For nested output types, the optimizer should aim to reduce the shredded components as much as possible before unshredding; however, the ability to estimate this in a distributed environment may not be straight-forward. For example, the SparkSQL performance from Section 7.3 suggested that optimizing for reduce shuffle does not always produce the best runtime overall. This was also observed in the pushing of local aggregates, which is a shuffle-based heuristic that was often not fully amortized and resulted in increased runtimes. The ability to apply relational optimization techniques to the framework will thus need to consider the factors related to distributed processing.

There is also the question of how to make a cost-based decision that occurs prior to plan construction. More specifically, how to cost shredding-specific transformations

such as materialization, the use of domains, and partial shredding. There are specific decisions to make within each transformation, as well as consider how one transformation impacts another. For example, large domains and shallow depths may exhibit better runtimes for the monolithic strategy in comparison to sequential. The implementation of domains can also be a factor in choosing a materialization strategy or when to use the domain-based optimization. Small domains could be broadcast, whereas large domains could benefit from local deduplication. Both of these decisions would also need to be made with respect to partially shredded inputs and queries. Support for partial shredding brings additional challenges, such as the question of when to unnest and when to operate locally on a nested collection stored within the parent tuple. All these shredding-specific decisions are not mutually exclusive and should be governed by a cost model that factors in distribution costs.

The shredding-specific transformations, however, are decisions made within the Shredding module, whereas costing happens at the plan level. The ability to cost shredding decisions will require enumerating the plans of all possible transformations. Heuristics that can reduce the plan space or the ability to access cost estimates in the Shredding module should be explored. Since partial shredding could require both flattening and shredding techniques, the ability to cost unnest, nest, and local operations on nested collections need to be developed. The number of optimizations that can be applied at both the language and plan level makes the costing of distributed nested data processing a key factor in the development of optimizations, as such this is a major area of ongoing work.

Biomedical applications and data science pipelines. This work has presented several use cases that highlight how the TraNCE framework can support biomedical analyses in research and clinical settings. Biological analysis platforms provide workflow languages, require manual optimization of each individual component, and are often highly application specific. TraNCE provides a high-level language that can employ several optimizations previously unexplored in the biomedical domain. Many improvements can be made to better support these analyses, such as upstream processing of raw genomics data and ease of interaction with external systems. Data access mechanisms could be provided that support ingesting of specialized biological data types in shredded form, which should be encompassed in the extensions to support a richer data model. Similar to pushing of feature selection filters (Chapter 9), advanced sta-

tistical calculations used in downstream applications could be pushed upstream into the distributed environment. These extensions should be further explored, in addition to evaluating the system with additional use cases and datasets.

The clinical exploration programs present an interesting perspective for the design of biomedical data integration infrastructure. Web-based data types are often nested, and flattening methods scale poorly. The ability of biomedical systems and analysis applications to work on the succinct, shredded representation presents interesting opportunities for optimization. For example, a frontend interface could display data sources in integrated format to the user while persisting the shredded representations in the backend. Subsequent requests for these data sources could access cached inputs to perform aggregate operations localized to specific levels, such as the likelihood measurements or risk scores. Future work should consider use cases where iterative exploration and aggregation occurs on the data, such as shared workloads within a medical institution. The clinical setting requires interfaces that support drag-and-drop mechanisms to construct more advanced queries on the backend, ie. a clinician will not write SQL, NRC, or in any other query language. The ability to build an interface that support such query construction for higher-level users is important to consider, and is relevant to the development of new frontends mentioned above.

Outside of clinical settings, consortium and data biobanks could consider using shredded representations in the backend. Datasets often occur as dump files, which have already gone through a pre-processing phase that employs flattening. Adapting these systems to use shredded representations could support the development of optimized data analysis pipelines.

The TraNCE framework should be further explored for opportunities to support the whole of the machine learning life cycle. For example, the data integration tasks of the burden-based use case were an initial look at how to support downstream learning analyses. Currently, sampling and cross-validation techniques are used in an effort to avoid constructing large-scale feature matrices. This could limit the signal of features and reduce model performance. TraNCE can scalably execute pre-processing steps to support inference on a whole dataset. Further, a system that supports a high-level language, large-scale data integration, and downstream learning tasks provides more opportunities for looking at optimizations that can be explored across the whole pipeline. The feature selection filters presented a set of micro-experiments for such op-

timizations, considering both runtime and model performance. The ability to unify the stages of a learning pipeline from data ingestion to model training and validation presents many opportunities to further explore optimizations.

Thesis Summary. Our work takes a step in exploring how components like shredded representations, query unnesting, and skew-resilient processing fit together to support scalable processing of nested collections. The biomedical applications of this work are used as a representative of modern analysis workloads. Though the results are anticipated to be applicable across many domains, it will be important to evaluate the system on additional use cases and datasets from other domains, such as web and finance. The work of this thesis has shown that there are many areas of nested data processing that should be re-evaluated with consideration to large-scale processing. In general, there are still many open areas of research for nested data, such as the interaction of nested data with distributed systems, query optimization, and cost estimation. All of these topics should be investigated further, especially given the prevalence of nested data in modern analyses. Many state-of-the-art systems today are leveraging techniques that have proven inefficient for processing nested collections. The use of a more succinct data representation should be considered. The TraNCE platform has promise in automating the challenges that arise for large-scale, distributed processing of nested collections. The shredding techniques presented throughout this thesis should be further evaluated with additional benchmarks and explored for use in existing systems in order to support scalable processing of nested collections.

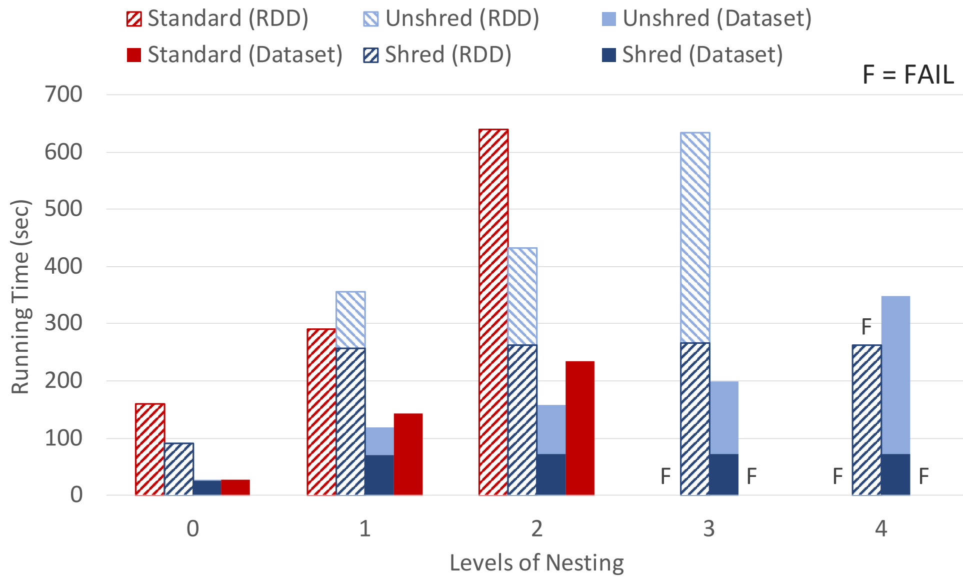
Appendix A

Appendices

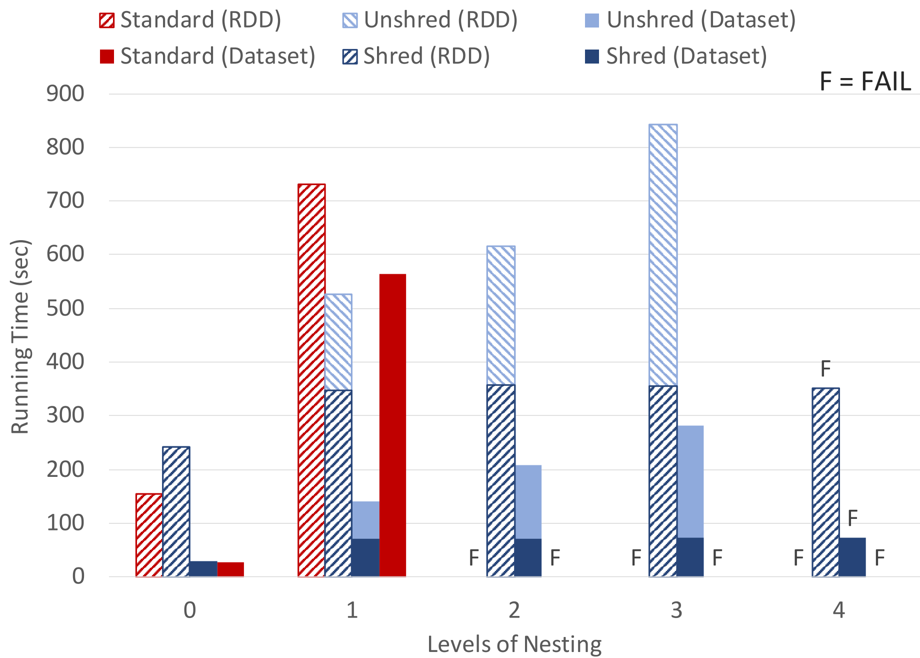
A.1 Spark RDDs vs Spark Datasets

The flat-to-nested and nested-to-nested queries are used to compare the performance of STANDARD (standard compilation route), SHRED (shredded compilation route without unshredding of output), and SHRED^{+U} (shredded compilation with unshredding) using RDDs and Datasets. Figure A.1 displays the results for the flat-to-nested queries with (A.1a) and without projections (A.1b). With projections, the results show that all strategies exhibit similar performance up to three levels of nesting. The strategies diverge at four levels of nesting where SHRED^{+U} and STANDARD with RDDs have a spike in total run time. The results without projections follow a similar trend, with strategies diverging earlier at lower levels of nesting. Without projections, SHRED^{+U} with RDDs shows increasingly worse performance starting at two levels of nesting. SHRED with RDDs and SHRED with Datasets have similar performance.

As stated in Section 7.1.1, the plan produced with unshredding in the shredded compilation and the plan for the standard compilation are identical; thus, the difference in performance is attributed to code generation for the unshredding procedure. Both methods use a series of cogroups to build up a nested set; however, unshredding requires additional map operations and intermediate object creation that are required for reconstructing nested objects from dictionaries. Case classes that lack binary encoders require a significant amount of time and space to create and store, which is a



(a) Narrow schema



(b) Wide schema

Figure A.1: Performance comparison of RDDs verse Datasets for the nested-to-nested benchmarked queries.

cost that only increases with the levels of nesting. SHRED^{+U} with RDDs grows exponentially as the number of nesting increases, bringing along the previous level with each level of nesting. This is also a problem for STANDARD, which sees worse performance with increasing levels of nesting but grows at a slower rate due to less object creation.

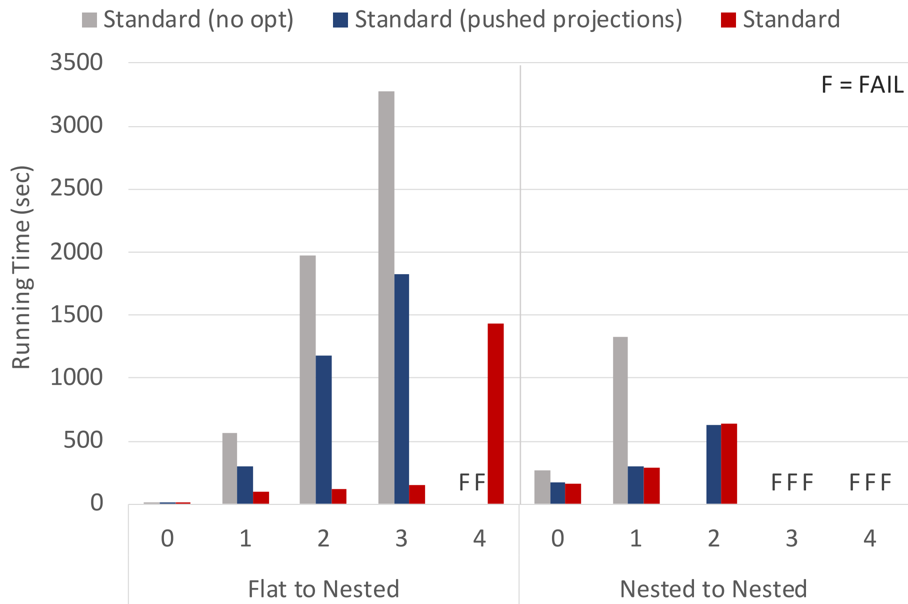
To consider what this means from a code generation perspective, compare the project operator of Figure 3.3 to the project operator in Figure 3.7. The RDD API maps over a relation and creates a new case class (`R`), whereas the Dataset API avoids this map and uses a select operator that accepts a series of attribute names. By explicitly stating the attributes, the Dataset API has delayed an explicit map operation allowing for further performance benefits from the Spark optimizer. Further, when the time comes to construct `R` objects, the Dataset API leverages the binary format for a much smaller memory footprint.

Figure A.1 further highlights the benefits of Datasets with nested-to-nested queries. Both with and without projections, the difference between SHRED with RDDs and SHRED with Datasets shows a 2× performance improvement of the explicit statement of attributes within the Dataset API, and STANDARD has decreased performance with RDDs. While the unnest operators used in the code generators both use flat-map operations, the Dataset API maintains a low-memory footprint for the newly created objects (`.as[R]` in the code generator).

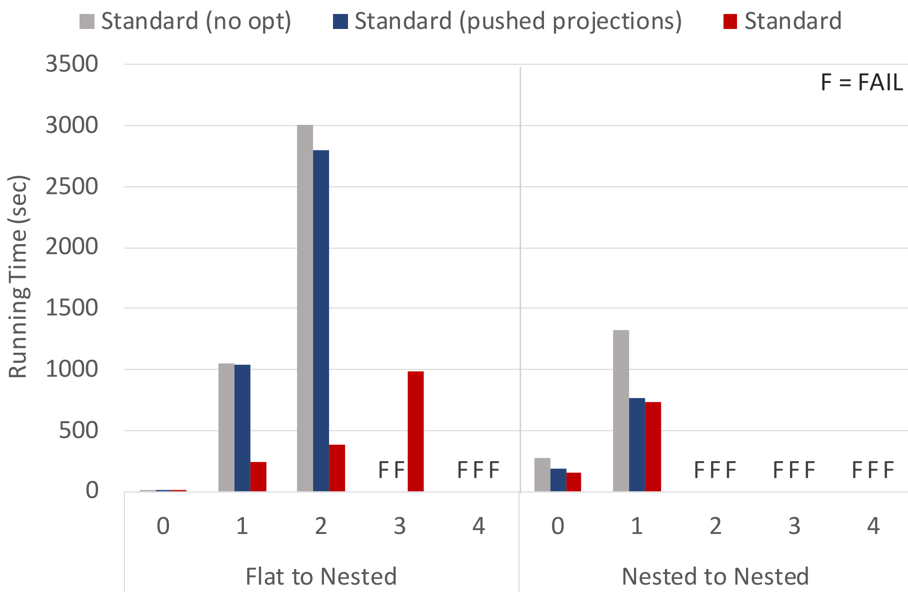
These results show that code generation with Datasets has minimized overhead in object creation and gains further improvements from passing meta-information to the Spark optimizer. Beyond the application to Spark, these results should be useful for further implementations of automated nested query processing on distributed systems.

A.2 Standard compilation framework optimizations

This experiment highlights how the framework can leverage database-style optimizations to automatically generate programs that are comparable to hand-optimized programs. Plans are generated using the standard compilation route with an increasing level of optimization applied to both the flat-to-nested and nested-to-nested queries.



(a) Narrow schema



(b) Wide schema

Figure A.2: Performance comparison of benchmarked queries for increasing optimization levels of the standard compilation route.

Figure A.2 shows the results of this experiment. Standard (no opt) is the standard compilation route with no optimizations; this corresponds to the plan that comes directly from the comprehension to plan translation. Standard (pushed projections) is the standard compilation route with projections pushed. Standard applies all optimizations that produce the optimal plan, which is the plan in all experiments in the body of the paper; this includes pushed projections, nesting operators pushed past join operators, and merged into cogroups where relevant.

The results show that even simple optimizations like pushing projections can provide major performance benefits for flattening methods. For example, Figure A.2a shows that projections have not only increased performance of the standard compilation route, but have allowed the strategy to survive to deeper levels of nesting. This is expected since the experiments in the previous sections have shown that the performance of Standard is heavily impacted by the presence of projections (ie. the number of attributes an output tuple). For nested-to-nested queries, Standard is the only strategy to survive past one level of nesting. These results show that database-style optimizations are not only beneficial to improve performance, but are necessary when using flattening methods even with shallow-nested objects.

The current state-of-the-art for defining programs over distributed, nested collections is hand-written code that has been manually optimized by an experienced developer. This is part of the programming mismatch described in Chapter 1. These results show that leveraging standard database-style optimizations can be useful in the automation of programs that operate on nested collections, and supports more systematic benchmarking of flattening methods than manually optimizing hand-written code.

A.3 One-vs-rest binary models for gene burdens based on raw counts

The accuracy and loss of the binary models from the one-vs-rest burden based analysis of Section 8.5 are presented in Figure A.3 through Figure A.8. Each model is presented for 10 epochs.

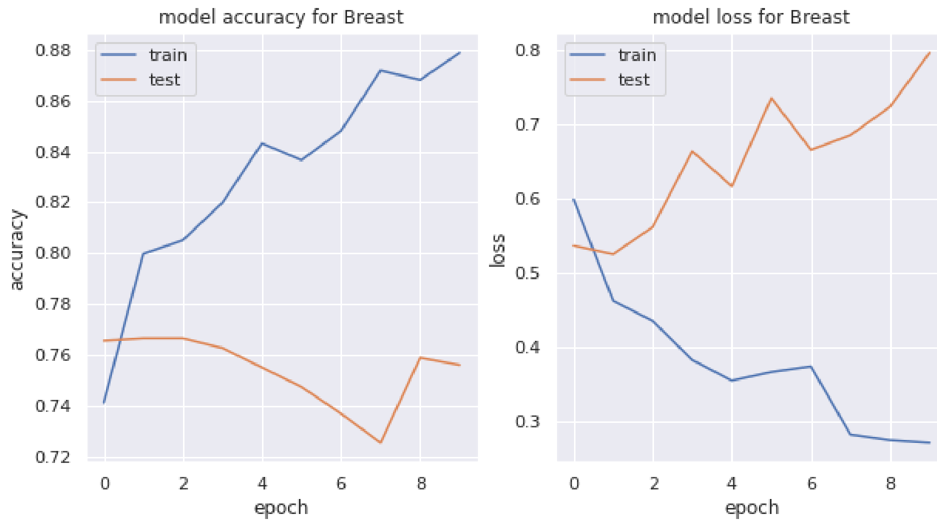


Figure A.3: The accuracy and loss of the binary neural network for breast.

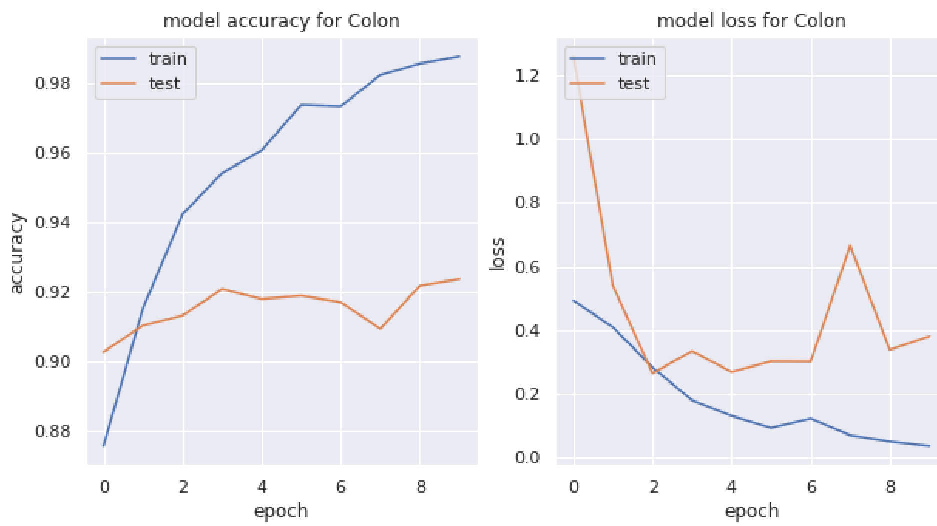


Figure A.4: The accuracy and loss of the binary neural network for colon.

Figure A.5: The accuracy and loss of the binary neural network for endometrial.

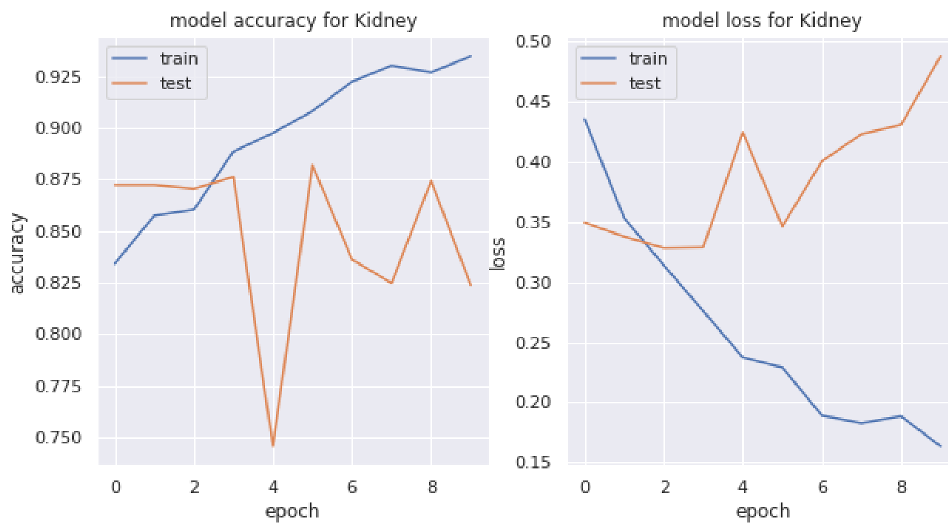


Figure A.6: The accuracy and loss of the binary neural network for kidney.

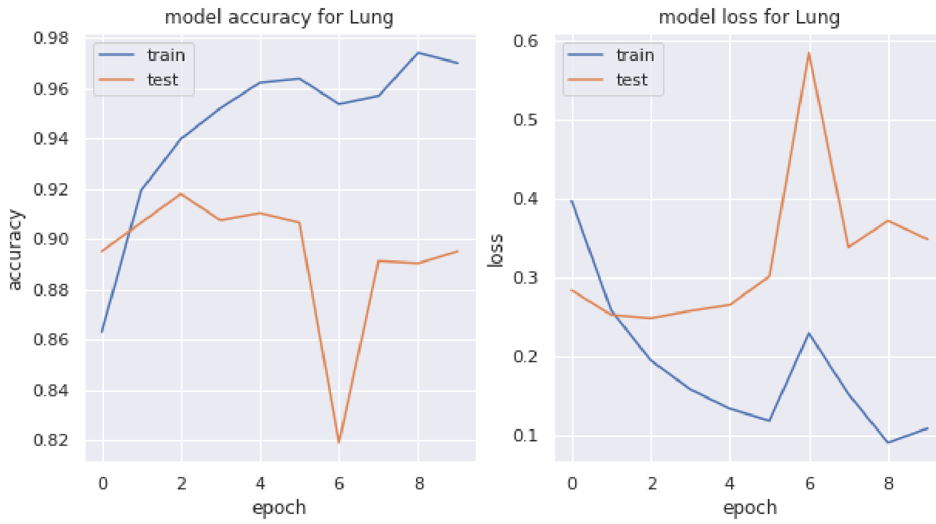


Figure A.7: The accuracy and loss of the binary neural network for lung.

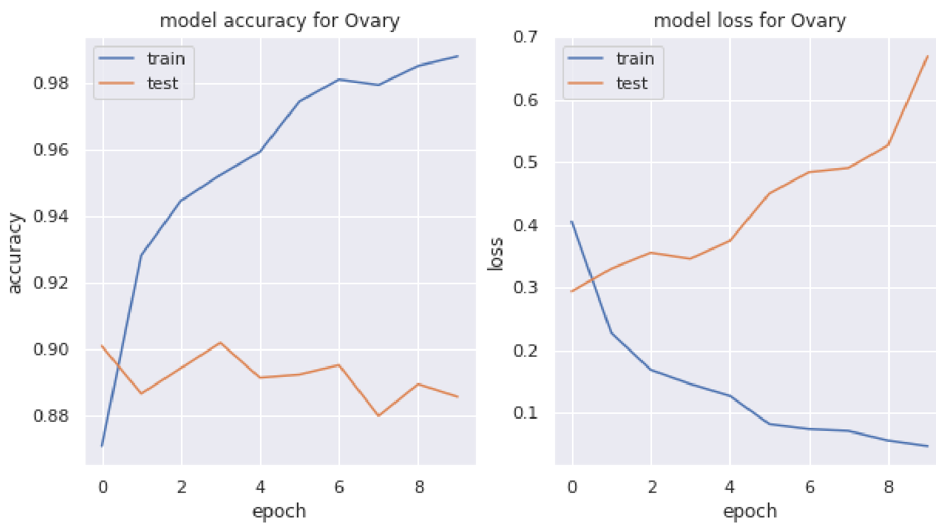


Figure A.8: The accuracy and loss of the binary neural network for ovary.

A.4 Feature selection experiments: preliminary exploration

These experiments describe preliminary explorations performed to motivate the experimental design. The first experiment determines how many features to return from feature selection. The second evaluates the pivoting functions from Scala (internal UDF) and Python (external UDF).

Number of features. To determine the number of features to pass to the predictor models `GeneImpactBurden`$^{\textsc{ChiSq}}$ and `GeneImpactBurden`$^{\text{ANOVA}}$ runs were performed set to return the top 20000 features. Figure A.9 and Figure A.10 displaying the loss and validation with a clear overfitting pattern.

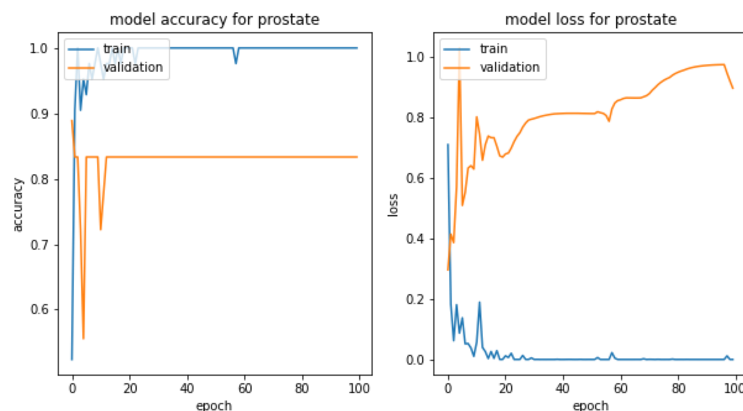


Figure A.9: Overfitting behavior in the training and validation error of the model using `GeneImpactBurden` and chi-square feature selection returning top 20000

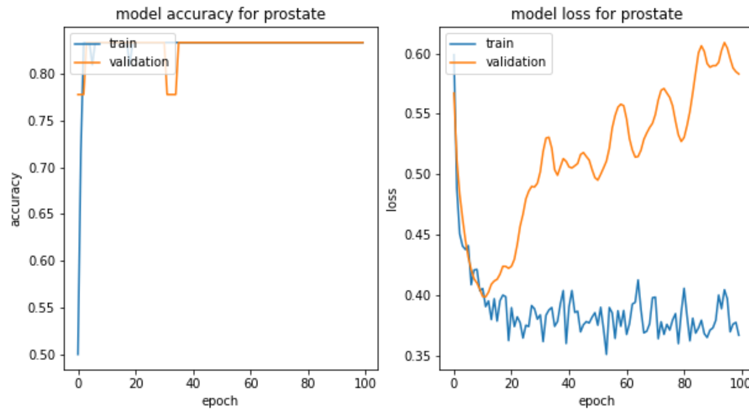


Figure A.10: Overfitting behavior in the training and validation error of the model using `GeneImpactBurden` and ANOVA feature selection returning top 20000.

Based on the confirmed overfitting for large number of features, a range of top features to return were explored using chi-square and ANOVA feature selection methods. The range starts at about 10% of the total features 5000, followed by 4000, 3000, 2000, 1000, 200, and 100. Models were then generated with 5-fold cross validation for 100 epochs. Table A.4 presents the testing accuracies for the $\texttt{GeneImpactBurden}^{\text{ChiSq}}$ and $\texttt{GeneImpactBurden}^{\text{ANOVA}}$ for each of the top feature sets returned - denoted in the column headers.

Each feature selection method chooses the top 200 features to use in the model. This was based on a preliminary run of the `GeneImpactBurden` program. The table below shows the results for other number of features returned. This is inline with the one-in-ten rule that says more than 10% of the features will overfit the model; larger numbers were shown to overfit whereas reduced number of features provides the best performance overall. The table shows that 200 top features returned provides the best performance; thus, the remaining experiments use top 200 from each feature selection method.

Method	**5000**	**4000**	**3000**	**2000**	**1000**	**200**	**100**
ChiSq	80%	80%	75%	80%	78%	**85%**	70%
ANOVA	80%	82%	82%	81%	84%	**88%**	72%

Table A.1: `GeneImpactBurden` accuracies by number of top features returned.

Pivot Evaluation. To motivate the use of Python-based UDFs for this case study, the performance of the Spark/Scala pivot function was evaluated against the Python counterpart. The `GeneImpactBurden` matrix was passed to the Spark/Scala pivot function and the runtime was evaluated for 600K total rows (all), as well as 500K, 400K, 300K, 200K, and 100K row subsets. The function was allowed to run for 5 minutes before it was killed. The Spark/Scala pivot function was unable to run to completion for any number of rows. The pivot in Pandas took 20 seconds for the full `GeneImpactBurden` matrix, which is 15x faster than the 5 minute cutoff time for the Spark/Scala function that never ran to completion. Given that pivot is an important operation in data science applications, the performance of this operation supports our decision to analyze Python-based UDFs for this case study.

Bibliography

[1] M. A. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” in *HotCloud*, 2010.

[2] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, “Nephele/-PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing,” in *Cloud Computing*, 2010.

[3] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] Apache Cassandra, 2020. cassandra.apache.org.

[5] Apache CouchDB, 2020. couchdb.apache.org.

[6] MongoDB, 2020. mongodb.com.

[7] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive Analysis of Web-Scale Datasets,” in *VLDB*, 2010.

[8] P. Buneman, S. Naqvi, V. Tannen, and L. Wong, “Principles of Programming with Complex Objects and Collection Types,” *Theoretical Computer Science*, vol. 149, no. 1, pp. 3–48, 1995.

[9] L. Wong, *Querying Nested Collections.* PhD thesis, Univ. Pennsylvania, 1994.

[10] L. Fegaras and D. Maier, “Optimizing Object Queries Using an Effective Calculus,” *ACM Transactions on Database Systems*, vol. 25, no. 4, pp. 457–516, 2000.

[11] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig Latin: A Not-so-Foreign Language for Data Processing,” in *SIGMOD*, 2008.

[12] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," in *VLDB*, 2008.

[13] Apache Hive, 2020. hive.apache.org.

[14] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, *et al.*, "F1: A Distributed SQL Database That Scales," in *VLDB*, 2013.

[15] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, *et al.*, "F1 Query: Declarative Querying at Scale," in *VLDB*, 2018.

[16] A. Alexandrov, A. Katsifodimos, G. Krastev, and V. Markl, "Implicit Parallelism through Deep Language Embedding," in *SIGMOD*, 2016.

[17] L. Fegaras and M. H. Noor, "Compile-Time Code Generation for Embedded Data-Intensive Query Languages," in *BigData Congress*, 2018.

[18] J. V. den Bussche, "Simulation of the Nested Relational Algebra by the Flat Relational Algebra," *Theor. Comput. Sci.*, vol. 254, no. 1-2, pp. 363–377, 2001.

[19] J. Cheney, S. Lindley, and P. Wadler, "Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets," in *SIGMOD*, 2014.

[20] C. Koch, D. Lupei, and V. Tannen, "Incremental View Maintenance For Collection Programming," in *PODS*, 2016.

[21] P. Beame, P. Koutris, and D. Suciu, "Skew in Parallel Query Processing," in *PODS*, 2014.

[22] M. A. H. Hassan, M. Bamha, and F. Loulergue, "Handling Data-skew Effects in Join Operations Using MapReduce," in *International Conference on Computational Science*, 2014.

[23] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling Data Skew in Parallel Joins in Shared-Nothing Systems," in *SIGMOD*, 2008.

[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the*

8th USENIX Conference on Operating Systems Design and Implementation, 2008.

[25] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[26] J. Smith, M. Benedikt, M. Nikolic, and A. Shaikhha, "Scalable Querying of Nested Data," in *VLDB*, 2021.

[27] J. Smith, M. Benedikt, M. Nikolic, and A. Shaikhha, "Scalable Querying of Nested Data," 2020. arxiv.org/abs/2011.06381.

[28] J. Smith, M. Benedikt, B. Moore, and M. Nikolic, "TraNCE: Transforming Nested Collections Efficiently," in *VLDB*, 2021.

[29] J. Smith, Y. Shi, M. Benedikt, and M. Nikolic, "Scalable Analysis of Multi-Modal Biomedical Data," *GigaScience*, vol. 10, 09 2021. giab058.

[30] TraNCE, 2020. github.com/jacmarjorie/trance.

[31] A. Makinouchi, "A Consideration on Normal Form of Not Necessarily Normalized Relation in the Relational Data Model," in *VLDB*, 1977.

[32] V. Breazu-Tannen, P. Buneman, and L. Wong, "Naturally embedded query languages," in *ICDT*, 1992.

[33] M. J. Carey and D. J. DeWitt, "Of Objects and Databases: A Decade of Turmoil," in *VLDB*, 1996.

[34] R. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, *The Object Database Standard: ODMG 2.0.* Morgan Kaufman, 1997.

[35] R. Cattell and D. Barry, *The Object Data Standard: ODMG 3.0.* Morgan Kaufman, 01 2000.

[36] A. Eisenberg and J. Melton, "SQL: 1999, Formerly Known as SQL3," in *SIGMOD*, 1999.

[37] L. Wong, "Kleisli, a Functional Query System," *Journal of Functional Programming*, vol. 10, no. 1, pp. 19–56, 2000.

[38] Apache Hadoop, 2020. hadoop.apache.org.

[39] 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 *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2016.

[40] 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 *SIGMOD*, 2015.

[41] J. Laskowski, *The Internals of Spark SQL*. Laskowski, Jacek, 2021.

[42] Apache Pig, 2020. pig.apache.org.

[43] A. Vitorovic, M. El Seidy, and C. Koch, "Load Balancing and Skew Resilience for Parallel Joins," in *ICDE*, 2016.

[44] S. B. Davidson, G. C. Overton, V. Tannen, and L. Wong, "BioKleisli: A Digital Library for Biomedical Researchers," *Int. J. on Digital Libraries*, vol. 1, no. 1, pp. 36–53, 1997.

[45] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "Links: Web Programming Without Tiers," in *Formal Methods for Components and Objects*, 2007.

[46] E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling Object, Relations and XML in the .NET Framework," in *SIGMOD*, 2006.

[47] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, *et al.*, "Spanner: Becoming a SQL system," in *SIGMOD*, 2017.

[48] F. N. Afrati, D. Delorey, M. Pasumansky, and J. D. Ullman, "Storing and Querying Tree-Structured Records in Dremel," in *VLDB*, 2014.

[49] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber, "FERRY: Database-Supported Program Execution," in *SIGMOD*, 2009.

[50] T. Grust, J. Rittinger, and T. Schreiber, "Avalanche-Safe LINQ Compilation," in *VLDB*, 2010.

[51] Citus, 2020. citusdata.com.

[52] L. Fegaras, "Compile-Time Query Optimization for Big Data Analytics," *Open Journal of Big Data (OJBD)*, vol. 5, no. 1, pp. 35–61, 2019.

[53] L. Fegaras, "An Algebra for Distributed Big Data Analytics," *Journal of Functional Programming*, vol. 27, 2017.

[54] Rumble, 2020. rumbledb.org.

[55] JSONiq, 2020. jsoniq.org.

[56] A. Rheinländer, U. Leser, and G. Graefe, "Optimization of Complex Dataflows with User-Defined Functions," *ACM Comput. Surv.*, vol. 50, May 2017.

[57] S. Chaudhuri and K. Shim, "Optimization of Queries with User-defined Predicates," *ACM Transactions on Database Systems*, vol. 24, 03 2000.

[58] D. Florescu, A. Levy, I. Manolescu, and D. Suciu, "Query Optimization in the Presence of Limited Access Patterns," in *SIGMOD*, 1999.

[59] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Li, W. Lin, J. Zhou, and L. Zhou, "Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions," *Proc. 9th USENIX Conf. Netw. Syst. Des. Implemen. (NSDI '12)*, 01 2012.

[60] J. Camacho-Rodríguez, D. Colazzo, I. Manolescu, and J. A. Naranjo, "PAX-Query: Parallel Analytical XML Processing," in *SIGMOD*, 2015.

[61] PySpark, 2020. spark.apache.org/docs/latest/api/python/index.html.

[62] scikit-learn, 2020. scikit-learn.org/stable.

[63] Keras, 2020. keras.io.

[64] J. Paredaens and D. Van Gucht, "Converting Nested Algebra Expressions into Flat Algebra Expressions," *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 65–93, 1992.

[65] E. Pasternak, R. Fenichel, and A. N. Marshall, "Tips for Creating a Block Language with Blockly," in *2017 IEEE Blocks and Beyond Workshop (B B)*, pp. 21–24, 2017.

[66] TCPH, "Transaction Processing Performance Council: TPC-H Benchmark," 2020. tpc.org.

[67] Zobra, 2016. github.com/zorba-processor/zorba.

[68] MonetDB, 2020. monetdb.org.

[69] Cockroach, 2020. github.com/cockroachdb/cockroach.

[70] VoltDB, 2020. voltdb.com.

[71] YugabyteDB, 2020. yugabyte.com.

[72] L. Cheng and S. Kotoulas, "Efficient Skew Handling for Outer Joins in a Cloud Computing Environment," *IEEE Transactions on Cloud Computing*, vol. 6, no. 2, pp. 558–571, 2018.

[73] W. Zhang and S.-L. Wang, "A Novel Method for Identifying the Potential Cancer Driver Genes Based on Molecular Data Integration," *Biochemical Genetics*, vol. 58, no. 1, pp. 16–39, 2020.

[74] N. R. C. U. Committee, "On the Nature of Biological Data," in *Catalyzing Inquiry at the Interface of Computing and Biology* (J. C. Wooley and H. S. Lin, eds.), ch. 3, National Academies Press (US), 2005.

[75] R. Hodson, "Precision Medicine," *Nature*, vol. 537, no. 7619, p. S49, 2016.

[76] K. He, D. Ge, and M. He, "Big Data Analytics for Genomic Medicine," *International Journal of Molecular Sciences*, vol. 18, p. 412, 02 2017.

[77] L. Coppola, A. Cianflone, A. M. Grimaldi, M. Incoronato, P. Bevilacqua, F. Messina, S. Baselice, A. Soricelli, P. Mirabelli, and M. Salvatore, "Biobanking in Health Care: Evolution and Future Directions," *Journal of Translational Medicine*, vol. 17, no. 1, p. 172, 2019.

[78] A. Auton, G. R. Abecasis, D. M. Altshuler, R. M. Durbin, D. R. Bentley, A. Chakravarti, Clark, *et al.*, "A Global Reference for Human Genetic Variation," *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.

[79] International Cancer Genome Consortium, 2020. icgc.org.

[80] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. M. Shaw, A. Brad, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart, "The Cancer Genome Atlas Pan-Cancer Analysis Project," *Nature Genetics*, vol. 45, no. 10, pp. 1113–1120, 2013.

[81] C. Sudlow, J. Gallacher, N. Allen, V. Beral, P. Burton, J. Danesh, P. Downey, P. Elliott, J. Green, M. Landray, B. Liu, P. Matthews, G. Ong, J. Pell, A. Silman, A. Young, T. Sprosen, T. Peakman, and R. Collins, "UK Biobank: An Open Access Resource for Identifying the Causes of a Wide Range of Complex Diseases of Middle and Old Age," *PLoS Medicine*, vol. 12, no. 3, pp. 1–10, 2015.

[82] J. Leipzig, "A Review of Bioinformatic Pipeline Frameworks," *Briefings in Bioinformatics*, vol. 18, no. 3, pp. 530–536, 2017.

[83] E. Afgan et al, "The Galaxy Platform for Accessible, Reproducible and Collaborative Biomedical Analyses: 2018 Update," *Nucleic Acids Research*, vol. 46, no. W1, pp. W537–W544, 2018.

[84] K. Voss, J. Gentry, and G. V. D. Auwera, "Full-Stack Genomics Pipelining with GATK4+ WDL+ Cromwell [version 1; Not Peer Reviewed]," *F1000Research*, p. 4, 2017.

[85] I. Curoverse, "Introduction to Arvados A Curoverse White Paper," tech. rep., Curoverse, Inc., 2014.

[86] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[87] M. Masseroli, P. Pinoli, F. Venco, A. Kaitoua, V. Jalili, F. Palluzzi, H. Muller, and S. Ceri, "GenoMetric Query Language: A Novel Approach to Large-scale Genomic Data Management," *Bioinformatics*, vol. 31, no. 12, pp. 1881–1888, 2015.

[88] Hail, 2015. github.com/hail-is/hail.

[89] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson, "ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing," Tech. Rep. UCB/EECS-2013-207, UCB/EECS, 2013.

[90] F. A. Nothaft, M. Massie, D. Timothy, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson, "Rethinking Data-Intensive Science Using Scalable Analytics Systems," in *SIGMOD*, 2015.

[91] Glow, 2019. github.com/projectglow/glow.

[92] K. Sato, "An Inside Look at Google BigQuery," *White Paper, Google Inc*, p. 12, 2012.

[93] D. Blakenberg, V. G. Kuster, E. Bouvier, D. Baker, E. Afgan, N. Stoler, G. Team, J. Taylor, and A. Nekrutenko, "Dissemination of Scientific Software with Galaxy ToolShed," *Genome Biology*, vol. 15, no. 2, p. 403, 2014.

[94] W. McLaren, L. Gil, S. E. Hunt, H. S. Riat, G. R. S. Ritchie, A. Thormann, P. Flicek, and F. Cunningham, "The Ensembl Variant Effect Predictor," *Genome Biology*, vol. 17, no. 1, p. 122, 2016.

[95] R. Vaser, S. Adusumalli, S. N. Leng, M. Sikic, and P. Ng, "SIFT Missense Predictions for Genomes," *Nature Protocols*, vol. 11, no. 1, p. 1073–1081, 2009.

[96] I. A. Adzhubei, S. Schmidt, L. Peshkin, V. E. Ramensky, A. Gerasimova, P. Bork, A. S. Kondrashov, and S. R. Sunyaev, "A Method and Server for Predicting Damaging Missense Mutations," *Nature Methods*, vol. 7, no. 4, pp. 248–249, 2010.

[97] K. Eilbeck, S. E. Lewis, C. J. Mungall, M. Yandell, L. Stein, R. Durbin, and M. Ashburner, "The Sequence Ontology: A Tool for the Unification of Genome Annotations," *Nature Methods*, vol. 6, p. R44, 2005.

[98] HTSJDK, "A Java API for High-Throughput Sequencing Data (HTS) Formats," 2020. samtools.github.io/htsjdk.

[99] D. Szklarczyk, A. Gable, D. Lyon, *et al.*, "STRING v11: Protein-Protein Association Networks with Increased Coverage, Supporting Functional Discovery in Genome-Wide Experimental Datasets," *Nucleic Acids Res*, vol. 47, no. D1, pp. D607–D613, 2019.

[100] T. Subramanian *et al.*, "Gene Set Enrichment Analysis: A Knowledge-Based Approach for Interpreting Genome-Wide Expression Profiles," *PNAS*, vol. 102, no. 43, pp. 15545–15550, 2005.

[101] L. Mootha *et al.*, "PGC-1alpha-Responsive Genes Involved in Xxidative Phosphorylation are Coordinately Downregulated in Human Diabetes," *Nature Genetics*, vol. 34, no. 3, pp. 267–273, 2003.

[102] D. Smedley, "The BioMart Community Portal: An Innovative Alternative to Large, Centralized Data Repositories," *Nucleic Acids Research*, vol. 43, pp. W589–W598, 04 2015.

[103] E. Birney, T. D. Andrews, P. Bevan, M. Caccamo, Y. Chen, *et al.*, "An Overview of Ensembl (GRCH37/Release 87)," *Genome Res.*, vol. 14, no. 5, pp. 925–928, 2004.

[104] i2b2, 2020. i2b2.org/software/index.html.

[105] M. Gabetta, I. Limongelli, E. Rizzo, A. Riva, D. Segagni, and R. Bellazzi, "BigQ: a NoSQL Based Framework to Handle Genomic Variants in i2b2," *BMC Bioinformatics*, vol. 16, no. 1, p. 415, 2015.

[106] S. N. Murphy, P. Avillach, R. Bellazzi, L. Phillips, M. Gabetta, A. Eran, M. T. McDuffie, and I. S. Kohane, "Combining Clinical and Genomics Queries Using i2b2 – Three Methods," *PLOS ONE*, vol. 12, pp. 1–16, 04 2017.

[107] J. M. Smith, M. Lathara, H. Wright, B. Hill, N. Ganapati, G. Srinivasa, and C. T. Denny, "Advancing Clinical Cohort Selection with Genomics Analysis on a Distributed Platform," *PLOS ONE*, vol. 15, pp. 1–20, 04 2020.

[108] Z. Z. Cheng F, Zhao J, "Advances in Computational Approaches for Prioritizing Driver Mutations and Significantly Mutated Genes in Cancer Genomes," *Briefings in Bioinformatics*, vol. 17, no. 4, pp. 642–656, 2016.

[109] S. Smemo, J. J. Tena, and M. A. Nóbrega, "Obesity-Associated Variants within FTO Form Long-Range Functional Connections with IRX3," *Nature*, vol. 507, no. 7492, p. 371–375, 2014.

[110] G. Kichaev, W. Y. Yang, S. Lindstrom, F. Hormozdiari, E. Eskin, A. L. Price, P. Kraft, and B. Pasaniuc, "Integrating Functional Data to Prioritize Causal Variants in Statistical Fine-Mapping Studies," *PLoS Genetics*, vol. 10, no. 10, 2014.

[111] L. Fancello, S. Gandini, P. G. Pelicci, and L. Mazzarella, "Tumor Mutational Burden Quantification from Targeted Gene Panels: Major Advancements and Challenges," *Journal for ImmunoTherapy of Cancer*, vol. 7, no. 1, p. 183, 2019.

[112] Z. R. Chalmers, C. F. Connelly, D. Fabrizio, L. Gay, S. M. Ali, R. Ennis, A. Schrock, B. Campbell, A. Shlien, J. Chmielecki, F. Huang, *et al.*, "Analysis of 100,000 Human Cancer Genomes Reveals the Landscape of Tumor Mutational Burden," *Genome Medicine*, vol. 9, no. 1, p. 34, 2017.

[113] W. Jiao, G. Atwal, P. Polak, R. Karlic, E. Cuppen, F. Al-Shahrour, P. J. Bailey, A. V. Biankin, P. C. Boutros, P. J. Campbell, D. K. Chang, *et al.*, "A Deep Learning System Accurately Classifies Primary and Metastatic Cancers using Passenger Mutation Patterns," *Nature Communications*, vol. 11, no. 1, p. 728, 2020.

[114] Y. Liang, H. Wang, J. Yang, X. Li, C. Dai, P. Shao, G. Tian, B. Wang, and Y. Wang, "A Deep Learning Framework to Predict Tumor Tissue-of-Origin Based on Copy Number Alteration," *Frontiers in Bioengineering and Biotechnology*, vol. 8, p. 701, 2020.

[115] Y. Zheng, Y. Ding, Q. Wang, Y. Sun, X. Teng, Q. Gao, W. Zhong, X. Lou, C. Xiao, C. Chen, Q. Xu, and N. Xu, "90-Gene Signature Assay for Tissue Origin Diagnosis of Brain Metastases," *Journal of Translational Medicine*, vol. 17, no. 1, p. 331, 2019.

[116] Q. Wang, M. Xu, Y. Sun, J. Chen, C. Chen, C. Qian, Y. Chen, L. Cao, Q. Xu, X. Du, and W. Yang, "Gene Expression Profiling for Diagnosis of Triple-Negative Breast Cancer: A Multicenter, Retrospective Cohort Study," *Front Oncol.*, vol. 9, p. 354, May 2019.

[117] J. Grewal, B. Tessier-Cloutier, M. Jones, S. Gakkhar, Y. Ma, R. Moore, A. Mungall, Y. Zhao, M. Taylor, K. Gelmon, H. Lim, D. Renouf, J. Laskin, M. Marra, S. Yip, and S. Jones, "Application of a Neural Network Whole Transcriptome-Based Pan-Cancer Method for Diagnosis of Primary and Metastatic Cancers," *JAMA Netw Open*, vol. 2, p. e192597, April 2019.

[118] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical Evaluation of Rectified Activations in Convolutional Network," *arXiv*, May 2015.

[119] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[120] M. Massie, et. al, "ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing," *UCB, Tech. Report UCB/EECS-2013*, vol. 207, 2013.

[121] X. Zhao, S. Guan, and K. L. Man, "An Output Grouping Based Approach to Multiclass Classification Using Support Vector Machines," in *Advanced Multimedia and Ubiquitous Engineering* (J. J. J. H. Park, H. Jin, Y.-S. Jeong, and M. K. Khan, eds.), (Singapore), pp. 389–395, Springer Singapore, 2016.

[122] P. H. Sudmant, T. Rausch, E. J. Gardner, R. E. Handsaker, A. Abyzov, J. Huddleston, Y. Zhang, K. Ye, G. Jun, M. Hsi-Yang Fritz, *et al.*, "An Integrated Map of Structural Variation in 2,504 Human Genomes," *Nature*, vol. 526, no. 7571, pp. 75–81, 2015.

[123] A. Subramanian, P. Tamayo, V. K. Mootha, S. Mukherjee, B. L. Ebert, M. A. Gillette, A. Paulovich, S. L. Pomeroy, T. R. Golub, E. S. Lander, *et al.*, "Gene Set Enrichment Analysis: A Knowledge-Based Approach for Interpreting Genome-Wide Expression Profiles," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 102, no. 43, pp. 15545–15550, 2005.

[124] J. Liu, C. Xu, W. Yang, Y. Shu, W. Zheng, and F. Zhou, "Multiple Similarly Effective Solutions Exist for Biomedical Feature Selection and Classification Problems," *Scientific Reports*, vol. 7, no. 1, p. 12830, 2017.

[125] Q. Zou, J. Zeng, L. Cao, and R. Ji, "A Novel Features Ranking Metric with Application to Scalable Visual and Bioinformatics Data Classification," *Neurocomputing*, vol. 173, pp. 346–354, 2016.

[126] R. J. Urbanowicz, R. S. Olson, P. Schmitt, M. Meeker, and J. H. Moore, "Benchmarking Relief-Based Feature Selection Methods for Bioinformatics Data Mining," *Journal of Biomedical Informatics*, vol. 85, pp. 168–188, 2018.

[127] A. Bommert, X. Sun, B. Bischl, J. Rahnenführer, and M. Lang, "Benchmark for Filter Methods for Feature Selection in High-Dimensional Classification Data," *Computational Statistics and Data Analysis*, vol. 143, p. 106839, 2020.

[128] R. J. Urbanowicz, M. Meeker, W. La Cava, R. S. Olson, and J. H. Moore, "Relief-Based Feature Selection: Introduction and Review," *Journal of Biomedical Informatics*, vol. 85, pp. 189–203, 2018.

[129] F. Neutatz, F. Biessmann, and Z. Abedjan, "Enforcing Constraints for Machine Learning Systems via Declarative Feature Selection: An Experimental Study," in *SIGMOD*, 2021.

[130] C. Zhang, A. Kumar, and C. Ré, "Materialization Optimizations for Feature Selection Workloads," *ACM Trans. Database Syst.*, vol. 41, Feb. 2016.

[131] Z. Chen, T. Gerke, V. Bird, and M. Prosperi, "Trends in Gene Expression Profiling for Prostate Cancer Risk Assessment: A Systematic Review," *Biomedicine Hub*, vol. 2, pp. 1–15, May 2017.

[132] O. S. Tătaru, M. D. Vartolomei, J. J. Rassweiler, O. Virgil, G. Lucarelli, F. Porpiglia, D. Amparore, M. Manfredi, G. Carrieri, U. Falagario, D. Terracciano, O. de Cobelli, G. M. Busetto, F. D. Giudice, and M. Ferro, "Artificial Intelligence and Machine Learning in Prostate Cancer Patient Management—Current Trends and Future Perspectives," *Diagnostics*, vol. 11, no. 2, 2021.

[133] WEB-based GEne SeT AnaLysis Toolkit, 2021. webgestalt.org.

[134] Y. Liao, J. Wang, E. J. Jaehnig, Z. Shi, and B. Zhang, "WebGestalt 2019: Gene Set Analysis Toolkit with Revamped UIs and APIs," *Nucleic Acids Research*, vol. 47, pp. W199–W205, 05 2019.

[135] D. A. Bader and S. E. McGuire, "Tumour Metabolism and its Unique Properties in Prostate Adenocarcinoma," *Nature Reviews Urology*, vol. 17, pp. 214–231, Apr 2020.