Scalable Querying of Nested Data

Citation for published version:

Digital Object Identifier (DOI):
10.14778/3430915.3430933

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
Proceedings of the VLDB Endowment (PVLDB)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Scalable Querying of Nested Data

Jaclyn Smith
Michael Benedikt
University of Oxford
first.last@cs.ox.ac.uk

Milos Nikolic
Amir Shaikhha
University of Edinburgh
first.last@ed.ac.uk

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, we propose a framework 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. We provide an extensive experimental evaluation, demonstrating significant improvements provided by the framework in diverse scenarios for nested collection programs.

PVLDB Reference Format:

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at github.com/jacmarjorie/trance.

1 INTRODUCTION

Large-scale, distributed data processing platforms such as Spark [46], Flink [8], and Hadoop [16] have become indispensable tools for modern data analysis. Their wide adoption stems from powerful functional-style APIs that allow programmers to express complex analytical tasks while abstracting distributed resources and data parallelism. These systems use an underlying 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 [3, 4, 33], 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 [32].

Despite natively supporting nested data, existing 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, local programs that use Scala collections often require non-trivial and error-prone translations to distributed Spark programs. Beyond programming difficulties, nested data processing requires scaling in the presence of large or skewed inner collections. We next elaborate these difficulties by means of an example.

Example 1. Consider a variant of the TPC-H database containing a flat relation Part with information about parts and a nested relation COP with information about customers, their orders, and the parts purchased in the orders. The relation COP stores customer names, order dates, part IDs and purchased quantities per order. The type of COP is:

```
Bag {{cname : string, corders : Bag {{odate : date, oparts : Bag { {pid : int, qty : real } } } } }
```

Consider a collection program 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 name. We can express this using nested relational calculus [10, 44] as:

```
for cop in COP union
{{
  (cname := cop.cname,
   corders :=
   for co in cop.corders union
     { {odate := coodate, 
       Q
         oparts := sumBy(pname) oparts
         for op in co.oparts union
           for p in Part union
             if op.pid == p.pid then
               Qcorders
               for p in Part union
                 { {pname := p.pname,
                   Qoparts
                   { {total := op.qty * p.price}}}
             }}
   )
}}
```

The program navigates to oparts in COP, joins each bag with Part, computes the amount spent per part, and aggregates using sumBy which sums up the total amount spent for each distinct part name. We next discuss the challenges associated with processing such examples using distributed frameworks.

Challenge 1: Programming mismatch. The translation of collection programs from local to distributed settings is not always straightforward. Distributed processing systems natively distribute collections only at the granularity of top-level tuples and provide no direct support for using nested loops over different distributed collections. To address these limitations, programmers resort to techniques that are either prohibitively expensive or error-prone. In
To address these challenges we advocate an approach that relies on three key aspects:

**Query Compilation.** When designing programs over nested data, programmers often first conceptualize desired transformations using high-level languages such as nested relational calculus [10, 44], monad comprehension calculus [21], or the collection constructs of functional programming languages (e.g., the Scala collections API). These languages lend themselves naturally to centralized execution, thus making them the weapon of choice for rapid prototyping and testing over small datasets.

To relieve programmers from manually rewriting program for scalable execution (Challenge 1), we propose using a compilation framework that automatically restructures programs to distributed settings. In this process, the framework applies optimizations that are often overlooked by programmers such as column pruning, selection pushdown, and pushing nesting and aggregation operators past joins. These may significantly cut communication overheads.

**Query and Data Shredding.** To achieve parallelism beyond top-level records (Challenge 2), we argue for processing over **shredded** data. In this processing model, nested collections are encoded as flat collections and nested collection queries are translated into relational queries [17, 44]. Prior work exploits the shredding transformation to implement nested relational query languages on top of commercial relational processors [13] and for incremental evaluation [28], both in a centralized setting.

The motivation for using shredding in distributed settings is twofold. First, shredding can enable full parallel processing of large nested collections and their even distribution among worker nodes. Second, for complex programs consisting of a sequence of nested-to-nested data transformations, shredding eliminates the need for the reconstruction of intermediate nested results. The shredded representation provides more opportunities for reduction of data through aggregation, and thus delays and minimizes the costs of the potentially expensive join and regrouping operations when computing the output.

In our example, the shredded form of COP consists of one flat top-level collection containing labels in place of inner collections and two dictionaries, one for each level of nesting, associating labels with flat collections. The top-level collection and dictionaries are distributed among worker nodes. We transform the example program to compute over the shredded form of COP and produce shredded output. The first two output levels are those from the shredded input. Computing the last output level requires joining one dictionary of COP with Part, followed by the sum aggregate; both operations are fully distributed. The shredded representation here eliminates the need for flattening operations and thus minimizes the amount of shuffled data. The shredded output can serve as input to another constituent query in a pipeline. If required downstream, the nested output can be restored by joining together the computed flat collections.

**Skew-Resilient Query Processing.** To handle data skew (Challenge 3), we propose using different evaluation plans for processing skewed and non-skewed portions of data. We transparently rewrite evaluation plans to avoid partitioning collections on values that appear frequently in the data, thus consequently avoid overloading few worker nodes with significantly more data. Our processing strategy deals with data skew appearing at the top level

Prior work has addressed some of these challenges. High-level scripting languages such as Pig Latin [34] and Scope [12] ease the programming of distributed data processing systems. Apache Hive [5], F1 [36, 38], Dremel [32], and BigQuery [37] provide SQL-like languages with extensions for querying nested structured data at scale. Emma [2] and DJQL [22] support for-comprehensions and respectively SQL-like syntax via DSLs deeply-embedded in Scala, and target several distributed processing frameworks. But alleviating the performance problems caused by skew and manually flattening nested collections remains an open problem.
of distributed collections, but when coupled with the shredding transformation, it can also effectively handle data skew increase, even when alternative methods are unable to complete at all.

2 BACKGROUND

We describe here our source language, which is based on nested relational calculus [10, 44], and our query plan language, based on the intermediate object algebra of [21].

Nested Relational Calculus (NRC). We support a rich collection programming language NRC as our source language. Programs are sequences of assignments of variables to expressions, where the expression language extends the standard nested relational calculus with primitives for aggregation and deduplication. Figure 1 gives the syntax of NRC. We work with a standard nested data model with typed data items. The NRC types are built up 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 \( T \). For ease of presentation, we restrict the bag content \( T \) to either be a tuple type or a scalar type, as shown in Figure 1. Set types are modeled as bags with multiplicity one. We refer to a bag of tuples with scalar attributes as a `@bag`.

RelOp is a comparison operator on scalars (e.g., \( =, \leq \)), PrimOp is a primitive function on scalars (e.g., +, *), and BoolOp is a boolean operator (e.g., \&\& or |). A simple term can be a constant, a variable, or an expression \( \theta.a \). Variables can be free, e.g., representing input objects, or can be introduced in for or let constructs. \( \theta \) takes the expression \( e \) and returns a singleton bag. Conversely, \( get(e) \) takes a singleton bag and returns its only element; if \( e \) is empty or has more than one element, get returns a default value. \( \emptyset \) returns an empty bag.

The if-then-else constructor for expressions of bag type can be expressed using the if-then and union constructors.

\[
\begin{align*}
& C ::= \text{Bag}(F) & \quad \text{Collection Type} \\
& F ::= \langle a_1 : T_1, \ldots, a_n : T_n \rangle & \quad \text{Flat Type} \\
& S ::= \text{int} \mid \text{real} \mid \text{string} \mid \text{bool} \mid \text{date} & \quad \text{Scalar Type}
\end{align*}
\]

Figure 1: Syntax of NRC.

In summary, we make the following contributions:

- We propose a compilation framework that transforms declarative programs over nested collections into plans that are executable in distributed environments.
- We provide transformations that enable the framework to compute over shredded relations. This includes an extension of the symbolic shredding approach, originating in [28], to a larger query language, along with a materialization phase for dealing with nested outputs.
- We introduce techniques that adapt generated plans to the presence of skew.
- We develop a micro-benchmark based on TPC-H and also a benchmark based on a real-world biomedical dataset, suitable for evaluating nesting collection programs. We show that our framework can scale as nesting and skew increase, even when alternative methods are unable to complete at all.

3 COMPILATION FRAMEWORK

Our framework transforms high-level programs into programs optimized for distributed execution. It can generate code operating directly over nested data, as well as code operating over shredded data. Figure 2 shows our architecture.

This section focuses on the standard compilation method, which uses a variant of the unnesting algorithm [21] to transform an input NRC program into an algebraic plan expressed in our plan language. A plan serves as input to the code generator producing executable code for a chosen target platform. For this paper, we consider Apache Spark as the target platform; other platforms such as Apache Flink, Cascading, and Apache Pig could also be supported.

The shredded compilation adds on a shredding phase that transforms an NRC program into a shredded NRC program that operates over flat collections with shredding-specific labels, dictionaries, and dictionary lookup operations. Section 4 describes the shredded data representation and the shredding algorithm.
then passes through the unnesting and code generation phases, as in the standard method.

Both compilation routes can produce skew-resilient code during the code generation phase. Our skew-resilient processing first uses a lightweight sampling mechanism to detect skew in data. Then, for each operator affected by data skew (e.g., joins), our framework generates different execution strategies that process non-skewed and skewed parts of the data. Section 5 describes our skew-resilient processing technique.

We describe here our standard compilation route for producing distributed programs from high-level queries. It relies on existing techniques from unnesting [21] and code-generation (e.g., [11]). It generates code that automatically inserts IDs and processes NULL values – techniques that play a role in the manually-crafted solutions discussed in the introduction – while adding optimizations and guaranteeing correctness. Further, this forms the basis for our shredded compilation route and skew-resilient processing module.

**Unnesting.** The unnesting stage translates each constituent query in an NRC program into a query plan consisting of the operators of the plan language, following the prior work [21], to facilitate code generation for batch processing frameworks.

The unnesting algorithm starts from the outermost level of a query and recursively builds up a subplan \( \textit{subplan}_k \) for each nested expression \( e \). The algorithm detects joins written as nested loops with equality conditions and also translates each \( \text{for} \) loop over a bag-typed attribute \( x.a \) to an unnest operator \( \mu^a \).

The algorithm enters a new nesting level when a tuple expression contains a bag expression. When at a non-root level, the algorithm generates the outer counterparts of joins and unnests. At each level, the algorithm maintains a set \( G \) of attributes used as a prefix in the key parameter of any generated nest operator at that level. The set \( G \) is initially empty. Before entering a new nesting level, the algorithm expands \( G \) to include the unique ID attribute and other output attributes from the current level.

The sumBy expression is translated to a sum aggregate \( \sum^+ \) with the key consisting of the grouping attributes from the first two levels and \( p.pname \) from the key attribute of the sumBy. The second-level subplan continues with a nest operation \( \Gamma^0 \) created from entering the second level at \( co\text{oparts} \) with grouping attributes \( \{copID,coID,cname,odate\} \), followed by another nest operation with grouping attributes \( \{copID,cname\} \) and a projection on \( \{cname,corders\} \).

The unnesting algorithm may produce suboptimal query plans. We optimize the plan from Figure 3 by projecting away unused attributes from the \( Part \) relation. We can push the sum aggregate \( \sum^+ \) past the join to compute partial sums of \( qty \) values over the output of \( \{co\text{oparts}\} \), grouped by \( \{copID,coID,cname,odate,pid\} \). However, a similar local aggregation over \( Part \) brings no benefit since \( pid \) is the primary key of \( Part \).

**Code Generation.** Code generation works bottom-up over a plan produced by the unnesting stage, translating an NRC program into a parallel collection program. For this paper, we describe our compilation in terms of the Spark API [46].

We model input/output bags as Spark Datasets, which are strongly-typed, immutable collections of objects; by default, collections are

![Figure 2: System architecture.](image)

![Figure 3: A query plan for the running example with output types. Bag types are annotated with \( B \).](image)
distributed at the granularity of top-level tuples. We translate operators of the plan language into operations over Datasets, taking into account that \( \Gamma^D \) and \( \Gamma^T \) also cast NULL values to the empty bag and respectively 0; implementation details can be found in [39].

The design choice to use Datasets was based on an initial set of experiments [39], which revealed that an alternative encoding – using RDDs of case classes – incurs much higher memory and processing overheads. Datasets use encoder objects to operate directly on the compact Tungsten binary format, thus significantly reducing memory footprint and serialization cost. Datasets also allow users to explicitly state which attributes are used in each operation, providing valuable meta-information to the Spark optimizer.

Operators affect 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 compilation route.

**Optimization.** The generated plan is subject to standard database optimizations; this includes pushing down selection, projection, and nest operators. Aggregations are pushed down when the key is known to be unique, based on schema information for inputs. Additional optimizations are applied during code generation; for example, a join followed by a nest operation is directly translated into a cogroup [39], 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. These optimizations collectively eliminate unnecessary data as early as possible and thus reduce data transfer during distributed execution.

**4 SHREDDED PIPELINE**

On top of the standard compilation route we build a shredded variant. We provide a novel approach that begins with an extension of symbolic query shredding from [28], applies a materialization phase that makes the resulting queries appropriate for bulk implementation, optimizes the resulting expressions, and then applies a custom translation into bulk operations. We refer to this variant as shredded comilation.

Our shredded representation encodes a nested bag \( b \) whose type \( T \) includes bag-valued attributes \( a_1, \ldots, a_k \) by a \( \text{Bag} \) \( b^F \) of type \( T^F \) where each \( a_i \) has a special type \( \text{Label} \) and an identifier of a lower-level bag. The representation of \( b \) also includes a dictionary tree \( b^D \) of type \( T^D \) capturing the association between labels and flat bags at each level.

The type \( T^F \) always a flat bag type, and the type \( T^D \), always a tuple type, are defined recursively. For a tuple type \( T = (a_1 : T_1, \ldots, a_n : T_n) \), \( T^F \) is formed by replacing each attribute \( a_i \) of bag type with a corresponding attribute of type \( \text{Label} \). \( T^D \) includes attributes \( a_i^{\text{fun}} \) and \( a_i^{\text{child}} \) for each bag-valued \( a_i \), where \( a_i^{\text{fun}} \) denotes the dictionary for \( a_i \) of type \( \text{Label} \rightarrow \text{Bag}(T_i^F) \), while \( a_i^{\text{child}} \) denotes the dictionary tree of type \( T_i^D \). Similarly, for a bag type \( T = \text{Bag}(T_1) \), we have \( T^F = \text{Bag}(T_1^F) \) and \( T^D = T^F \); for scalar types, we have \( T^F = T \) and \( T^D = \{} \).

**Example 3.** Based on the type of COP (Example 1), the shredded representation of COP consists of a top-level flat bag \( \text{Bag} \) of type \( \text{Bag}(\{\text{name} : \text{string}, \text{corders} : \text{Label}\}) \) and a dictionary tree \( \text{COP}^D \) of tuple type

\[
\begin{aligned}
\text{corders}^{\text{fun}} &: \text{Label} \rightarrow \text{Bag}((\text{odate} : \text{date}, \text{oparts} : \text{Label})), \\
\text{corders}^{\text{child}} &: \text{Bag}((\text{oparts}^{\text{fun}} : \text{Label} \rightarrow \text{Bag}((\text{pid} : \text{string}, \text{qty} : \text{real}), \text{oparts}^{\text{child}} : \text{Bag}(()))) \}
\end{aligned}
\]

The COPD dictionary tree encodes the `corders^{fun}` dictionary for corders labels, the `oparts^{fun}` dictionary for oparts labels, and the nesting structure via the child attributes. Since the type system prevents nesting tuples inside tuples, each child dictionary is wrapped in a singleton bag.

We can convert nested objects to their shredded representations and vice versa. A value shredding function 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 performs the opposite conversion [28]. High-level definitions of the shredding and unshredding of a value of type \( T \) are straightforward, defined by induction on \( T \).

**Shredded NRC.** Our goal in query shredding is to convert a source NRC program to work over shredded representations of input and output. As an intermediate stage in shredding, we move to a symbolic representation for output dictionaries, defined as partial functions from labels to bags. We will use an intermediate query language, denoted NRC\text{Lb+\lambda}, which extends NRC with a new atomic type for labels and a function type for dictionaries. The grammar of NRC\text{Lb+\lambda} is:

\[
\begin{aligned}
\text{e} &::= \text{[Similar to Figure 1]} \\
| \text{NewLabel(var, ...) | match e =NewLabel(var, ...) then e} \\
| \text{match e =MatLookup(e, e) | lambda.e} \\
| \text{@DictTreeUnion e} \\
T &::= \text{[Similar to Figure 1]} \\
| \text{Label} &::= \text{Label Type} \\
| \text{Label} &\rightarrow \text{Bag}(F) &::= \text{Dictionary Type} \\
\end{aligned}
\]

The `NewLabel(x_1, ..., x_n)` construct creates a new label encapsulating the values of variables \( x_1, ..., x_n \) of flat types. To deconstruct labels we have a "label matching construct":

\[
\begin{aligned}
\text{match l =NewLabel(x) then F(x, y)}
\end{aligned}
\]

where \( F \) is a bag expression, and \( x \) and \( y \) are tuples of variables. Formally, given any label \( l \) and a binding for \( y \), we find the unique \( x \) such that \( l = NewLabel(x) \) and evaluate \( F \). If there is no such \( x \), \( F \) returns the empty bag. In this expression, \( x \) becomes bound although it is free in \( F \).

We have the standard \( \lambda \) abstraction restricted to label parameters: if \( e \) is an expression of type \( T \) and \( l \) is a label variable, then we can form \( \lambda l.e \) of type \( \text{Label} \rightarrow T \). We also have the standard function application: if \( \theta_1 \) is an expression of type \( \text{Label} \rightarrow T \) and \( \theta_2 \) an expression of type \( \text{Label} \), then we can form \( \text{Lookup}(\theta_1, \theta_2) \) of type
T. MatLookup(e₁, e₂) corresponds to a lookup of the label returned by e₂ within the bag of label-bags pairs returned by e₁. Finally, we have a variation of the binary union for expressions representing dictionary trees, denoted as DictTreeUnion.

**Query Shredding Algorithm.** A query shredding algorithm takes as input an expression e of type T and produces an expression eₐ of type Tᴰ and a dictionary tree eᵢ of tuple type Tᵢᴰ. The algorithm transforms the evaluation problem for e to evaluation problems for eₐ and eᵢ. When the output of e on an input i is a₀, then the outputs of eₐ and eᵢ on the shredded representation of i will be eₐ and eᵢ, respectively.

Our shredding transformation consists of two phases. We proceed first with a **symbolic query shredding** phase, producing succinct expressions manipulating intermediate and output dictionaries defined via λ expressions. This phase adapts the work from [28] to our source language. This is followed by a **materialization transformation** producing a sequence of expressions free of λ abstractions.

Figure 4 shows the shredding transformation via recursive functions F and D; the full translation can be found in [39]. Given a source expression e, the invocation F(e) returns the expression eₐ computing the flat version of the output, while D(e) returns the dictionary tree eᵢ corresponding to eₐ.

The base cases are those of constants and variables: for a scalar c, we have F(c) = c and D(c) is an empty dictionary tree (line 1); for a variable x of type T, the two functions return x and x² whose types Tᵀ and Tᵢᴰ depend on T as described above (line 2).

The interesting cases are those of tuple construction and tuple projection. For each bag-valued attribute aᵢ in a tuple constructor (line 3), F replaces the bag expression eᵢ with a new label encapsulating the (flat) free variables of eᵢ. The dictionary tree of the tuple constructor includes two attributes: aᵢᵀFunctional, which represents the mapping from such a label to the flat variant eᵢᵀ of eᵢ; and aᵢ chapId, which represents the dictionary tree for eᵢ. To conform with the type system, each child dictionary tree is wrapped as a singleton tuple constructor includes two attributes:

<table>
<thead>
<tr>
<th>eᵢ</th>
<th>eᵢᵀ</th>
<th>eᵢ ChapId</th>
</tr>
</thead>
<tbody>
<tr>
<td>aᵢᵀ</td>
<td>aᵢ ChapId</td>
<td></td>
</tr>
</tbody>
</table>

We match Q to the for construct in F (line 8) and recur to derive Qᵢ. For co ∈ COP (a symbolic dictionary), for co in Lookup(co.ctors, co.child) union

```
  (odate := co. date, oparts := NewLabel(co))
```

We omit the unused let binding to co in the expression computing the dictionary corders. The query producing the lowest-level dictionary tree QᵢCorders is:

```
  let co := get(co.ctors, co.child) in
    (oparts := λl. match l = NewLabel(co) then
      sumBy(COP): for op in Lookup(op.ctors, op.child) union
        for pᵢ in Part_i union
         if (pᵢ. pid == op. pid) then
           ((pname := pᵢ.pname, total := op.qty * pᵢ.price)),
         opartschild := ({}))
```

Our implementation [40] refines the above shredding algorithm to return only the relevant attributes of free variables in a NewLabel; for instance, the corder’s labels in QF need to capture only copᵀ.functional.corder’s labels but not copᵀ.functional.values.

**Materialization Algorithm.** The symbolic dictionaries produced by the symbolic query shredding algorithm keep the inductively-formed expressions succinct. The second phase of the shredding process, which we refer to as **materialization**, eliminates λ terms in favor of expressions that produce an explicit representation of the shredded output.

The materialization phase produces a sequence of assignments, given a shredded expression and its dictionary tree. For each symbolic dictionary QᵢDict, the dictionary tree, the transformation creates one assignment MatDictᵢDict ≜ e, where the expression e computes a bag of tuples representing the materialized form of QᵢDict. 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 MatDictᵢDict 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 QᵢDictᵢ for each label.

The materialization algorithm from Figure 5 produces a sequence of assignments given a shredded expression, its dictionary tree, and
a variable to represent the top-level expression. The \texttt{Materialize}
procedure first replaces the (input) symbolic dictionaries in $\mathcal{F}$ 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 1st binding produced by the shredding algorithm (line 3).

The \texttt{MaterializeDict} procedure performs a depth-first traversal
of the dictionary tree. For each label-valued attribute $a$, we first emit an assignment computing the set of labels produced in the
parent assignment (lines 3-4). We then rewrite the dictionary $\mathcal{A}$
to replace all references to symbolic dictionaries, each of them guaranteed to have a matching assignment since the traversal is
top-down. We finally produce an assignment computing a bag of
label-value tuples representing the materialized form of $\mathcal{A}$ (lines
6-8), before recurring to the child dictionary tree (line 9).

\textbf{Example 5}. We showcase the \texttt{Materialize} procedure on the
shredded queries $Q^F$ and $Q^D$ from Example 4. Let $\mathcal{MAT}$,$\mathcal{MAT}$,$\mathcal{LD}$
and $\mathcal{LD}$ denote the materializations of the top-level input bag $\mathcal{COP}$ and
the two symbolic dictionaries from $\mathcal{COP}$, respectively. The procedure replaces $\mathcal{COP}$ by $\mathcal{MAT}$ in $Q^F$ (line
1), followed by emitting the assignment to the variable TopBag:

$$\begin{align*}
\text{TopBag} & \leftarrow \text{for } \mathcal{COP}^F \text{ in } \mathcal{MAT} \text{ union} \\
& \{\langle \text{name} := \mathcal{COP}^F . \text{name}, \text{corders} := \text{NewLabel}(\mathcal{COP}^F) \rangle \}
\end{align*}$$

\texttt{MaterializeDict} produces an assignment computing the set of
colders labels from the parent TopBag (lines 3-4):

$$\begin{align*}
\text{LabDomain}_{\text{corders}} & \leftarrow \\
& \text{dup}(\text{for } x \text{ in } \text{TopBag union} \{\langle \text{label} := x . \text{corders} \rangle \})
\end{align*}$$

The function uses the labels from \texttt{LabDomain}_{\text{corders}} to materialize
the corders\texttt{Fun} dictionary (lines 6-8):

$$\begin{align*}
\text{MatDict}_{\text{corders}} & \leftarrow \\
& \text{for } l \text{ in } \text{LabDomain}_{\text{corders}} \text{ union} \\
& \{\langle \text{label} := l . \text{label}, \\
& \text{value} := \text{match } l . \text{label} = \text{NewLabel}(\mathcal{COP}^F) \text{ then} \\
& \text{for } \mathcal{COP}^F \text{ in } \text{MatLookup}(\text{MatCOP}_{\text{corders}}, \mathcal{COP}^F . \text{corders}) \\
& \text{union } \{\langle \text{odate} := \mathcal{COP}^F . \text{odate}, \text{oparts} := \text{NewLabel}(\mathcal{COP}^D) \rangle \}
\end{align*}$$

The query computing value corresponds to the body of the 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 oparts\texttt{Fun} from \texttt{MatDict}_{\text{corders}}:

$$\begin{align*}
\text{LabDomain}_{\text{oparts}} & \leftarrow \\
& \text{dup}(\text{for } x \text{ in } \text{MatDict}_{\text{corders}} \text{ union } \{\langle \text{label} := x . \text{oparts} \rangle \})
\end{align*}$$

\texttt{MatDict}_{\text{oparts}} produces an assignment computing the set of
opparts labels from the parent LabDomain (lines 3-4):

$$\begin{align*}
\text{LabDomain}_{\text{oparts}} & \leftarrow \\
& \text{dup}(\text{for } x \text{ in } \text{LabDomain}_{\text{oparts}} \text{ union} \{\langle \text{label} := x . \text{oparts} \rangle \})
\end{align*}$$

The first rule recognizes that a child symbolic dictionary is an
expression of the form:

$$\text{match } l . \text{label} = \text{NewLabel}(x) \text{ then}$$

$$\text{for } y \text{ in } \text{Lookup}(D, x, a) \text{ union } e$$

where the only used attribute of $x$ is $a$ of label type. We can skip
computing the domain of labels for this dictionary and compute

\begin{figure}
\centering
\begin{tabular}{|c|c|c|}
\hline
Pattern of $e$ & $\mathcal{F}(e)$ & $\mathcal{D}(e)$ \\
\hline
1 $c$ & $c$ & $\langle \rangle$ \\
2 $\var{e}$ & $\var{e}$ & $\var{e}$ \\
3 $\langle ... \ a_i := e_i, ... \ e \rangle$ & $\langle ... \ a_i := \text{NewLabel}(x^F) \text{ if } x \in \text{FreeV ars} \ (e_i) \rangle$ & $\langle ... \ a_i := \lambda l . \text{match } l = \text{NewLabel}(...) \text{ then } \mathcal{F}(e_i), \ a_i^\text{child} := \{D(e_i)\}, ... \rangle$ \\
4 $\langle ... \ a_j := e_j, ... \ e \rangle$ & $\langle ... \ a_j := \mathcal{F}(e_j), ... \rangle$ & $\langle ... \ \rangle$ \\
5 $\langle e . a \rangle$ & $\text{Lookup}(D(e), a^\text{fun}, \mathcal{F}(e), a)$ & $\text{get}(D(e), a^\text{child})$ \\
6 $\langle e . a \rangle$ & $\mathcal{F}(e), a$ & $\langle \rangle$ \\
7 $\langle \rangle$ & $\langle \rangle$ & $\langle \rangle$ \\
8 $\langle for \ a \ in \ e_1 \ \text{union} \ \theta_2 \rangle$ & $\langle let \ \var{D} := D(e_1) \text{ in} \ \\
\quad \text{for } \var{F} \text{ in } \mathcal{F}(e_1) \ \text{union} \ \mathcal{F}(e_2) \rangle$ & $\langle let \ \var{D} := D(e_1) \text{ in } D(e_2) \rangle$ \\
9 $\langle e_1 \ \text{or} \ e_2 \rangle$ & $\mathcal{F}(e_1) \ \text{or} \ \mathcal{F}(e_2)$ & $\mathcal{D}(e_1) \ \text{DictTreeUnion} \ \mathcal{D}(e_2)$ \\
10 $\langle \mathcal{F}(e) \rangle$ & $\mathcal{F}(e)$ & $\mathcal{D}(e)$ \\
11 $\langle \mathcal{F}(e_1), \mathcal{F}(e_2) \rangle$ & $\mathcal{F}(e_1), \mathcal{F}(e_2)$ & $\langle \rangle$
\hline
\end{tabular}
\caption{Query shredding algorithm.}
\end{figure}
We benefit from this rule when the size of the label domain from the label-value pairs directly from the materialized dictionary. The second rule for domain elimination recognizes when a label appearance is comparable to the size of the label-value pairs directly from the constructed variable assignment is 

\[ \text{MaterializeDict}(e^D, \text{variable ParentIndex}) \]

1. `let e^D = \langle a_1 := e_1, \ldots, a_n := e_n \rangle`
2. `foreach a^un \in \langle a_1, \ldots, a_n \rangle`
3. `MaterializeDict(y^D, \text{variable ParentIndex})`
4. `for l in \text{LabDomain}_{\text{index}, a} \union`
5. `let z := (l.a) in`
6. `for y in z.value \union e`}

We benefit from this rule when the size of the label domain from the parent assignment is comparable to the size of the materialized dictionary. 

Example 6. Returning to Example 5, applying this rule avoids producing `LabDomain_{corders}` as an intermediate result. The materialization of `corders_{\text{union}}` now corresponds to:

\[ \text{MaterializeCorders}_{\text{union}} \]

1. `for z in \text{MatCOP}_{\text{corders, oparts}} union`
2. `\langle \text{label} := z.label,
3. value := let x := (a := z.label) in
4. for y in z.value \union e \rangle}

We can extend this rule to match a `sumBy` construct around the `for` construct, which also enables computing `MatDict_{corders, oparts}` directly from `MatCOP_{corders, oparts, corders_oparts}`

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

\[
\lambda l. \text{match } l = \text{NewLabel}(x) \text{ then }
\]

\[
\text{for } y \text{ in } Y \text{ union if } (y.a == x.b) \text{ then } e
\]

If no other attribute of `x` appears in `e`, we can produce the label-value pairs from `Y` using the value of `y.a`:

\[
\text{groupBy}_{\text{label}} \text{for } y \text{ in } Y \text{ union }
\]

\[
\text{let } x := (b := y.a) \text{ in }
\]

\[
\text{\{ \langle \text{label} := \text{NewLabel}(x), \text{value} := e \rangle \}}
\]

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

Extensions for Shredded Compilation. Given that the output of materialization is an NRC program with expressions containing only a subset of NRC\(_{\text{labels}} + \lambda\), there are only a few extensions required to support compilation for shredded data. Dictionaries are simply considered to be a special instance of a bag, keyed by values of a label type and supporting lookup operations. A `MatLookup` is translated directly into an outer join in the plan language, providing opportunities for pushed aggregation and merging with nest operators to form cgroups.

The shredded representation allows nested operations in the input query to be translated to operations working only on the dictionaries relevant to that level; we refer to such operations as localized operations. For example, the `sumBy` in the running example will operate directly over the `oparts` dictionary. This produces a plan, similar to a subset of the plan in Figure 3, that has isolated the join and aggregate operation to the lowest level dictionary. This avoids flattening the input via the series of unnest operators and avoids the programming mismatch described in Section 4.

The label-bag representation of dictionaries can lead to overheads when we want to join with a bag value; thus, a relational dictionary representation of label-value pairs can be useful for implementation. We introduce an operator `BagToDict` to provide flexibility in dictionary representation, which casts a bag to a dictionary while delaying explicit representation until the relevant stage of compilation.

For our current code generation, dictionaries are represented the same as bags, `Dataset[T]` where `T` contains a label column (label) as key. Top-level bags that have not been altered by an operator have no partitioning guarantee and are distributed by the same default strategy as bags in the standard compilation route. Dictionaries have a label-based partitioning guarantee; the key-based partitioning guarantee described above, where all values associated to the same label reside on the same partition.

5 SKEW-RESILIENT PROCESSING

We provide a novel approach to skew handling, which adapts a generated plan based on statistics of the data at runtime and supports shredding. The framework up to this point has not addressed skew-related bottlenecks. Skew is a consequence of key-based partitioning where all values with the same key are sent to the same partition. If the collected values of a key (key1) are much larger than the collected values of another key (key2), then the partition housing `key1` values will take longer to compute than others. We refer to such keys as heavy, and all others as light. In the worst case, the partition is completely saturated and the evaluation is heavily burdened by reading and writing to disk. We mitigate the effects of skew by implementing skew-aware plan operators that automate the skew-handling process.

The identification of heavy keys is central to automating the skew-handling process. We use a sampling procedure to determine heavy keys in a distributed bag, considering a key heavy when at least a certain threshold of the sampled tuples (10%) in a partition are associated with that key; in our experiments, we use the threshold of 2.5%. Based on the heavy keys, the input bag is split into a light component and a heavy component. We refer to these three components (light bag, heavy bag, and heavy keys) as a skew-triple.
A skew-aware operator is a plan operator that accepts and returns a skew-triple. If the set of heavy keys is not known, then the light and heavy components are unioned and a heavy key set is generated. This set of heavy keys remains associated to that skew-triple until the operator does something to alter the key, such as a join operation.

In essence, the skew-aware plan consists of two plans, one that works on the light component (light plan), and one that works on the heavy component (heavy plan). The light plan follows the standard implementation for all operators. The light plan ensures that values corresponding to light keys reside in the same partition. The heavy plan works on the heavy input following an alternative implementation. The heavy plan ensures the distribution of values associated to heavy keys. This ability to preserve the partitioning guarantee of light keys and ensure the distribution of large collections associated to heavy keys enables skew-resilient processing.

Figure 6 provides the implementation of the main skew-aware operations. 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 \( \frac{100}{3} = 40 \) 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. Broadcast operations maintain skew-resilience by ensuring the heavy key values in the larger relation remain at their current location and are not consolidated to the same node.

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 \( \Gamma^+ \) mitigates skew-effects by default by reducing the values of all keys. A grouping operation \( \Gamma^\circ \) cannot avoid skew-related bottlenecks. More importantly, skew-handling for nested output types would be detrimental to our design – attempting 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 compilation route, 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 6.

<table>
<thead>
<tr>
<th>Plan Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma^{agg} \text{ value} \ X )</td>
<td>val (X_L, X_H, hk) = X.heavyKeys(f)</td>
</tr>
<tr>
<td>( X \triangleright f(x) = g(y) \ Y )</td>
<td>val Y_L = Y.filter(y =&gt; !hk(g(y)))</td>
</tr>
<tr>
<td>( \text{BagToDict} \ X )</td>
<td>val light = X_L.repartition(x =&gt; x.label)</td>
</tr>
</tbody>
</table>

Figure 6: Skew-aware implementation for the plan language operators using Spark Datasets.

Our experimental results can be summarized as follows:

- For flat-to-nested queries, shredded compilation adds no overhead when producing nested results and shows up to 25x gain when producing shredded results.
- For nested input, the shredded compilation has a 16x improvement for shallow nesting and scales to deeper levels which flattening methods cannot handle.
- The shredded representation provides more opportunities for optimizations in nested-to-flat queries with aggregation, providing a 6x improvement over the flattening methods.
- Skew-aware shredded compilation outperforms flattening methods with 33x reduction in shuffling for moderate skew, as well as graceful handling of increasing amounts of skew while the flattening methods, skew-aware and skew-unaware, are unable to complete at all.

Full details of experimental evaluation can be found in [39].

**Experimental environment.** 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. We provide summary information on shuffling cost, with full details provided in [39]. All missing values correspond to a run that crashed due to memory saturation of a node.

**Benchmarks.** We create two NRC query benchmarks; a microbenchmark based on the standard TPC-H schema and the second based on biomedical datasets from the International Cancer Genome Consortium (ICGC) [26]. Our TPC-H benchmark contains a suite of flat-to-nested, nested-to-nested, and 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 the Lineitem table at level 0, then group across Orders, Customer, Nation, then Region, as the level increases. Each query has a wide variant where we keep all the attributes, and a narrow variant which follows the grouping with a projection at each level. For scale factor 100, this organization gives query results with 600 million, 150 million, 15 million, 25, and 5 top-level tuples. Flat-to-nested queries perform the iterative grouping above to the relational inputs, returning a nested output. At the lowest level we keep the partkey and quantity of a Lineitem. At the higher levels the narrow variant keeps only a single attribute, e.g., order_date for Orders, customer_name for Customer, etc. The nested-to-nested queries take the materialized result of the flat-to-nested queries as input and perform a join with Part at the lowest level, followed by sumBy price as in Example 1. The nested-to-nested queries thus produce the same hierarchy as the flat-to-nested queries. Finally, the nested-to-at queries follow the same construction as the nested-to-nested queries, but apply sumBy name at top-level, where name is one of the top-level attributes; this returns a flat collection persisting only attributes from the outermost level. We use the skewed TPC-H data generator [43] to generate 100GB datasets with a Zipfian distribution. Skew factor 4 gives the greatest skew, of external competitors: an implementation via encoding in SparkSQL [6]; Citus, a distributed version of Postgres [14]; MongoDB [33], and the recently-developed nested relational engine DiQL [20]. Since SparkSQL outperformed the other competitors, this section continues with a comparison to only SparkSQL. Full discussion of additional competitors can be found in [39]. SparkSQL does not support explode (i.e., UNNEST) operations in the SELECT clause, requiring the operator to be kept with the source relation which forces flattening for queries that take nested input. The SparkSQL queries were manually written based on this restriction. Standard denotes the standard compilation and produces results that match the type of the input query. Shred denotes the shredded variant of our system with domain elimination, leaving its output in shredded form. Unshred represents the time required to unshred the materialized dictionaries, returning results that match the type of the input query. Unshred is often considered in combination with Shred to denote total runtime of shredded compilation when the output type is nested. Shred assumes a downstream operation can consume materialized dictionaries. The above strategies do not implement skew-handling; we use Standard, Shred, and Shred to denote the skew-handling variations. We execute the optimal plan associated to a given query, which at a minimum includes pushing projections and domain-elimination (Section 4); further optimizations described within the context of each experiment.

**Flat-to-nested queries for non-skewed data.** The TPC-H flat-to-nested queries are used to explore building nested structures from flat inputs when the data is not skewed. Figure 7 shows the running times of SparkSQL, Standard, Shred, and Unshred for increasing levels of nesting.

Shred runs to completion for all levels, remaining constant after the first level and exhibiting nearly identical runtimes and max data shuffle for narrow and wide. Unshred and Standard have comparable runtimes overall and max data shuffle that is 20x that of Shred. For deeper levels of nesting, these methods fail to store the local nested collections when tuples are wide. SparkSQL does not perform the cogroup optimization (Section 3) and is unable to scale for the small number of top-level tuples that occur with deeper levels of nesting even when tuples are narrow. Shred shows 6x improvement for narrow queries and 26x for wide queries. The flat-to-nested case introduces a worst case for shredding, regrouping everything up in the unshredding phase without reduction. Even in this case, shredded compilation adds no overhead in comparison to flattening.

![Figure 7: Performance comparison of the narrow and wide benchmarked TPC-H queries with varying levels of nesting (0-4).](image-url)

**Evaluation strategies and competitors.** We compare the standard compilation (Section 3) to the shredded one (Section 4). On each of our benchmarks, we have also compared to a wide array of external competitors: an implementation via encoding in SparkSQL [6]; Citus, a distributed version of Postgres [14]; MongoDB [33], and the recently-developed nested relational engine DiQL [20].
Nested-to-nested queries for non-skewed data. The TPC-H nested-to-nested queries are used to explore the effects of aggregation on non-skewed data. We use the wide version of the flat-to-nested queries as input to evaluate the impact of projections on nested input. Figure 7 shows the runtimes of SparkSQL, STANDARD, Shred, and Unshred for increasing levels of nesting. SparkSQL exhibits consistently worse performance while maintaining the same total amount of shuffled memory as STANDARD. This behavior starts for 0 levels of nesting, suggesting overheads related to pushing projections that may be compounded when combined with the explode operator at deeper levels of nesting. SparkSQL does not survive even for one-level of nesting for wide tuples. The rest of the methods diverge at one level of nesting, with Shred exhibiting the best performance overall.

For wide queries STANDARD is burdened by the top-level attributes that must be included in sumBy {pname, qty, price}, and only finishes for one level of nesting. In the narrow case, STANDARD survives the aggregation but fails for the small number of top-level tuples in the deeper levels of nesting. Unshred shows a linear drop in performance as the number of top-level tuples decrease from 150 million to 25 tuples, but at a much lower rate compared to the flattening methods. With 5 top-level tuples, Unshred suffers from a poor distribution strategy that must maintain the wide tuples in nested local collections, and crashes due to memory saturation. Overall, Shred and Unshred show up to a 8x performance advantage for shallow levels of nesting and exhibit 3x less total shuffle. Further, in comparison to the previous results without aggregation, the localized aggregation produces up to a 3x performance gain in unshredding. This experiment highlights how the succinct representation of dictionaries is a key factor in achieving scalability. The use of this succinct representation enables localized aggregation (Section 4) which avoids data duplication, reduces the size of dictionaries, and lowers unshredding costs to support processing at deeper levels of nesting even when the returning nested output.

Nested-to-flat with non-skewed inputs. We use nested-to-flat queries to explore effects of aggregation for queries that navigate over multiple levels of the input; such queries are challenging for shredding-based approaches due to an increase in the number of joins in the query plan. Figure 7 shows the runtimes for SparkSQL, STANDARD, and Shred, where the unshredding cost for flat outputs is zero. Similar to the nested-to-nested results, STANDARD is unable to run for the small number of top-level tuples associated with deeper levels of nesting for both the narrow and wide case. SparkSQL has the worst performance overall, and cannot complete for even one level of nesting when tuples are wide.

Shred outperforms STANDARD with a 6x runtime advantage, over a 2x total shuffle advantage for wide queries, and runs to completion even when STANDARD fails to perform at all. 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, which reduces the first-level dictionary and results in a cheap join operation to reassociate the next two levels. 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.

Due to the nature of the non-skewed TPC-H data, STANDARD does not benefit from local aggregation (before unnesting) and also does not benefit from pushing the aggregation past the nested join since the top-level information must be included in the aggregation key. These results show how the shredded representation can introduce more opportunities for optimization in comparison to traditional flattening methods, supporting more cases where pushed and localized operations can be beneficial.

Skew-handling. We use the narrow variant of the nested-to-nested queries with two levels of nesting to evaluate our skew-handling procedure. We use the materialized flat-to-nested narrow query (COP) as input.

The TPC-H data generator produces skew by duplicating values; thus, pushing aggregations reduces the duplicated values associated to a heavy key and diminishes skew. We found that skew-aware methods benefit from aggregation pushing, whereas skew-aware methods benefit more from maintaining the distribution of heavy keys (skew-resilience, Section 5). We also found [39] benefits to skew-aware methods when aggregations are not pushed.

Figure 8 shows runtimes for each method for increasing amount of skew, pushing aggregation for only skew-unaware methods. ShredSkew has up to a 15x performance gain over other methods for moderate amounts of skew. The skew join of ShredSkew shuffles up to 33x less for moderate skew and up to 74x less for high skew than the skew-unaware join of Shred, which has first performed aggregation pushing. At greater amounts of skew, SparkSQL, Standard, and ShredSkew crash due to memory saturation from flattening skewed inner collections.

The performance gain of skew-aware shredded compilation is attributed to the succinctness of dictionaries, which provides better support for the skew-aware operators. The shredded compilation route, regardless of skew-handling, runs to completion for all skew factors. Beyond supporting the distribution of the large-skewed inner collections, the localized aggregation reduces the size of the lower-level dictionary thereby decreasing the skew. This highlights how the shredded representation is able to handle skew even before the skew-handling procedure is applied. Overall, we see that the
skew-aware components ads additional benefits by maintaining skew-resilience and reducing the amount of data shuffle required to handle heavy keys.

**Biomedical Benchmark.** Figure 9 shows the runtimes for Spark-SQL, STANDARD, and SHRED for the E2E pipeline of the biomedical benchmark. The final output of E2E is flat, so no unshredding is needed. Since STANDARD and SparkSQL fail during STEP2, we also provide results with a smaller dataset using only 6GB BN2, 2GB BF1, and 4GB BF2. SHRED exhibits many advantages, overall surviving the whole pipeline. The results for STEP1 using the small and full dataset exhibit similar behavior to the nested-to-flat results for TPC-H. Since the whole BN2 input is flattened and there are many projections during STEP1, STANDARD and SparkSQL are not too burdened by the nested joins that occur while flattening.

The methods diverge at STEP2, where a nested join between BN1 and the output of STEP1 leads to an explosion in the amount of data being shuffled. SHRED displays up to a 16x performance gain for STEP2 with the small dataset. Despite the small number of output attributes, the result of the join between flattened output of STEP1 and BN1 produces over 16 billion tuples and shuffles up to 2.1TB before crashing. The results for SHRED highlight that this is an expensive join, the longest running time for the whole pipeline; however, the succinct dictionary representation allows for a localized join at the first level of the output of STEP1, reducing the join to 10 billion tuples and the shuffling to only 470GB. Thus we avoid carrying around redundant information from the top-level and avoid the outer-join required to properly reconstruct the top-level tuples.

The E2E queries thus exhibit how a succinct representation of dictionaries is vital for executing the whole of the pipeline, with a 7x advantage for SHRED on the small dataset and scaling to larger datasets when the flattening methods are unable to perform. The queries also highlight how an aggregation pipeline that eventually returns flat output can leverage the succinct representation of dictionaries without the need for reassociating the dictionaries.

**7 RELATED WORK**

Declarative querying against complex objects has been investigated for decades in different contexts, from stand-alone implementations on top of a functional language, as in Kleisli [45], language-integrated querying approaches such as Links [15] and LINQ [31]. Alternative approaches [2] give more fine-grained programming abstractions, integrating into a host language like Scala.

The use of flat representations dates to early foundational studies of the nested relational model [17, 44]. Implementation of this idea in the form of query shredding has been less investigated, but it has been utilized by Cheney et al. [13] in the Links system and Grust et al. [24, 25] in the Ferry system. Both these systems focus on the generation of SQL queries. Koch et al. [28] provide a shredding algorithm in order to reduce incremental evaluation of nested queries to incremental evaluation of flat queries. They do not deal with scalable evaluation or target any concrete processing platform. However, our symbolic shredding transformation is closely related to the one provided by [28], differing primarily in that [28] supports only queries returning bag type without aggregation.

Query unnesting [21, 27], on the other hand, deals with the programming model mismatch, both for flat queries and nested ones. Fegaras and Maier’s unnesting algorithm [21] de-correlates standard NRC queries to support a more efficient bulk implementation, exploiting the richness of the object data model. In the process they introduce an attractive calculus for manipulating and optimizing nested queries. A number of recent applications of nested data models build on this calculus. For example, CleanM [23] uses it as the frontend language for data cleaning and generates Spark code. Similarly, DQL [20] and MQL [19] provide embedded DSLs in Scala generating efficient Spark and Flink programs with query unnesting and normalization. These works do not deal with limitations of standard nested representations, nor do they provide support for skew handling.

Google’s analytics systems such as Spanner [7], F1 [36, 38], and Dremel [32] support querying against complex objects. Dremel performs evaluation over a “semi-flattened” [1] format in order to avoid the space inefficiencies caused by fully flattening data. Skew-resilience and query processing performance are not discussed in [1], which focuses on the impact on storage, while details of the query-processing techniques applied in the commercial systems are proprietary. Skew-resilience in parallel processing [9, 29], and methods for efficient identification of heavy keys [35] have been investigated for relational data. But for nested data the only related work we know of targets parallel processing on a low-level parallel language [41], rather than current frameworks like Spark.

**8 CONCLUSION**

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. Our results show that the platform has promise in automating the challenges that arise for large-scale, distributed processing of nested collections; showing scalable performance for deeper levels of nesting and skewed collections even when state-of-the-art flattening methods are unable to perform at all. In the process we have developed both a micro-benchmark and a benchmark based on a real-world, biomedical use case. A number of components of a full solution still remain to be explored. A crucial issue, and a target of our ongoing work, is cost estimation for these programs, and the application of such estimates to optimization decisions within the compilation framework.
REFERENCES


[7] David F Bacon, Nathan Baler, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Files, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, and others. 2017. Spanner: Becoming a SQL system. In SIGMOD.


[34] Christopher Olston, Benjamin Reed, Ulfarvarivastavara, Riva Kumar, and Andrew Tomkins. 2008. Pig Latin: a not-so-foreign language for data processing. In SIGMOD.


