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:
Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. Scalable Querying of Nested Data. PVLDB, 14(3): 445-457, 2021.
doi:10.14778/3430915.3430933

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].

Proceedings of the VLDB Endowment, Vol. 14, No. 3 ISSN 2150-8097.
doi:10.14778/3430915.3430933

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 := co.odate,
Q   |    |   oparts := sumBy^total_pname(
|   |    |             | for op in co.oparts union
| Q_corders            |  for p in Part union
|   |    |             |   if op.pid == p.pid then
|   |    |        Q_oparts    {⟨ pname := p.pname,
|   |    |             |        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

our example, a common error is to try to iterate over Part, which is a collection distributed across worker nodes, from within the map function over COP that navigates to oparts, which are collections local to one worker node. Since Part is distributed across worker nodes, it cannot be referenced inside another distributed collection. Faced with this impossibility, programmers could either replicate Part to each worker node, which can be too expensive, or rewrite to use an explicit join operator, which requires flattening COP to bring pid values needed for the join to the top level. A natural way to flatten COP is by pairing each cname with every tuple from corders and subsequently with every tuple from oparts; however, the resulting (cname, odate, pid, qty) tuples encode incomplete information – e.g., omit customers with no orders – which can eventually cause an incorrect result. Manually rewriting such a computation while preserving its correctness is non-trivial.

Challenge 2: Data representation. Default top-level partitioning can lead to poor scalability with nested data, particularly with few top-level tuples and/or large inner collections. The number of top-level tuples limits the level of parallelism by requiring all data on lower levels to persist on the same node, which is detrimental regardless of the size of these inner collections. Conversely, large inner collections can overload worker nodes and increase data transfer costs, even with larger numbers of top-level tuples. In our example, a small number of customers in COP may lead to poor cluster utilization because few worker nodes process the inner collections of COP. To cope with limited parallelism, an alternative is to flatten nested data and redistribute processing over more worker nodes. In addition to the programming challenges, unnesting collections leads to data duplication and consequently redundant computation. Wide flattened tuples increase memory pressure and the amount of data shuffled among the worker nodes. Alleviating problems such as disk spillage and load imbalance requires rewriting of programs on an ad hoc basis, without a principled approach.

Challenge 3: Data skew. Data skew can significantly degrade performance of large-scale data processing, even when dealing with flat data only. Skew can lead to load imbalance where some workers perform significantly more work than others, prolonging run times on platforms with synchronous execution such as Spark, Flink, and Hadoop. The presence of nested data only exacerbates the problem of load imbalance: inner collections may have skewed cardinalities – e.g., very few customers can have very many orders – and the standard top-level partitioning places each inner collection entirely on a single worker node. Moreover, programmers must ensure that inner collections do not grow large enough to cause disk spill or crash worker nodes due to insufficient memory. □

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 DIQL [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.

Our approach. 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

$$
\begin{aligned}
\mathsf{P} &::= (\mathsf{var} \Leftarrow \mathsf{e})^* \\
\mathsf{e} &::= \emptyset_{\mathsf{Bag}\,(F)} \mid \{\mathsf{e}\} \mid \mathtt{get}(\mathsf{e}) \\
&\quad \mid c \mid \mathsf{var} \mid \mathsf{e}.a \mid \langle a_1 := \mathsf{e}, \ldots, a_n := \mathsf{e} \rangle \\
&\quad \mid \mathtt{for}\ \mathsf{var}\ \mathtt{in}\ \mathsf{e}\ \mathtt{union}\ \mathsf{e} \mid \mathsf{e} \uplus \mathsf{e} \\
&\quad \mid \mathtt{let}\ \mathsf{var} := \mathsf{e}\ \mathtt{in}\ \mathsf{e} \mid \mathsf{e}\ \mathit{PrimOp}\ \mathsf{e} \\
&\quad \mid \mathtt{if}\ \mathsf{cond}\ \mathtt{then}\ \mathsf{e} \mid \mathtt{dedup}(\mathsf{e}) \\
&\quad \mid \mathtt{groupBy}_{\mathsf{key}}(\mathsf{e}) \mid \mathtt{sumBy}^{\mathsf{value}}_{\mathsf{key}}(\mathsf{e}) \\
\mathsf{cond} &::= \mathsf{e}\ \mathit{RelOp}\ \mathsf{e} \mid \neg\mathsf{cond} \mid \mathsf{cond}\ \mathit{BoolOp}\ \mathsf{cond} \\
T &::= S \mid C \\
C &::= \mathsf{Bag}\,(F) && \text{– Collection Type} \\
F &::= \langle a_1 : T, \ldots, a_n : T \rangle \mid S && \text{– Flat Type} \\
S &::= \mathsf{int} \mid \mathsf{real} \mid \mathsf{string} \mid \mathsf{bool} \mid \mathsf{date} && \text{– Scalar Type}
\end{aligned}
$$

Figure 1: Syntax of NRC.

of distributed collections, but when coupled with the shredding transformation, it can also effectively handle data skew present in inner collections.

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-word 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.

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 $\mathsf{Bag}\,(T)$. For ease of presentation, we restrict the bag content T to be either 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 flat bag .

RelOp is a comparison operator on scalars (e.g., ==, ≤), *PrimOp* is a primitive function on scalars (e.g., +, ∗), and *BoolOp* is a boolean operator (e.g., &&, ||). A simple term can be a constant, a variable, or an expression e.a. Variables can be free, e.g. representing input objects, or can be introduced in `for` or `let` constructs. {e} 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. $\emptyset_{\mathsf{Bag}\,(F)}$ returns an empty bag.

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

`dedup`(e) takes the bag e and returns a bag with the same elements, but with all multiplicities changed to one. We impose a restriction that will be useful in our query shredding method (Section 4): the input to `dedup` must be a flat bag.

$\mathtt{groupBy}_{key}$(e) groups the tuples of bag e by a collection of attributes key (in the figure we assume a single attribute for simplicity) and for each distinct values of key, produces a bag named GROUP containing the tuples from e with the key value, projected on the remaining attributes.

$\mathtt{sumBy}^{\mathsf{value}}_{\mathsf{key}}$(e) groups the tuples of bag e by the values of their key attributes and for each distinct value of key, sums up the attributes value of the tuples with the key value.

In both `groupBy` and `sumBy` operators, we restrict the grouping attributes key to be flat.

Plan Language. Our framework also makes use of a language with algebraic operators, variants of those described in [21]. Our language includes selection σ, projection π, join $\bowtie$, and left outer join $⟕$, as in 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 $⟖\mu^a$ is a variant of μ^a that in addition extends each tuple from the outer bag with a unique ID and pairs such tuples with `NULL` values when a is the empty bag.

The nest operator $\Gamma^{\mathsf{agg}}_{key}{}^{value}$ defines a key-based reduce, parameterized by the aggregate function agg, which could be addition (+) or bag union ($\uplus$). This operator also casts `NULL` values introduced by the outer operators into 0 for the + aggregate and the empty bag for the $\uplus$ aggregate.

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. The shredded program

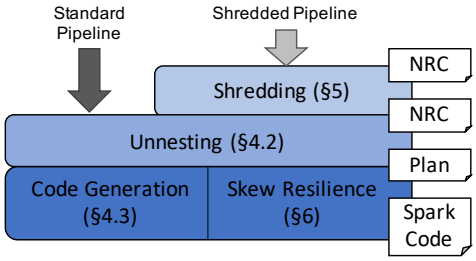


Figure 2: System architecture.

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 $subplan^{\mathsf{e}}$ for each nested expression e. The algorithm detects joins written as nested loops with equality conditions and also translates each `for` loop over a bag-typed attribute $x.a$ to an unnest operator μ^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 $\mathcal{G}$ of attributes used as a prefix in the key parameter of any generated nest operator at that level. The set $\mathcal{G}$ is initially empty. Before entering a new nesting level, the algorithm expands $\mathcal{G}$ to include the unique ID attribute and other output attributes from the current level. The $\mathtt{sumBy}^{\mathsf{value}}_{\mathsf{key}}(\mathsf{e})$ operator translates to $\Gamma^{+}{}^{\mathsf{value}}_{\mathcal{G},\mathsf{key}}(subplan^{\mathsf{e}})$, while $\mathtt{groupBy}_{\mathsf{key}}(\mathsf{e})$ translates to $\Gamma^{\uplus}{}^{\mathsf{value}}_{\mathcal{G},\mathsf{key}}(subplan^{\mathsf{e}})$, where *value* represents the remaining non-key attributes in e. These Γ operators connect directly to the subplan built up from unnesting e.

The example below details the plans produced by the unnesting algorithm on the running example. We will see that while unnesting addresses the programming mismatch, it highlights the challenges introduced by flattening mentioned in the introduction.

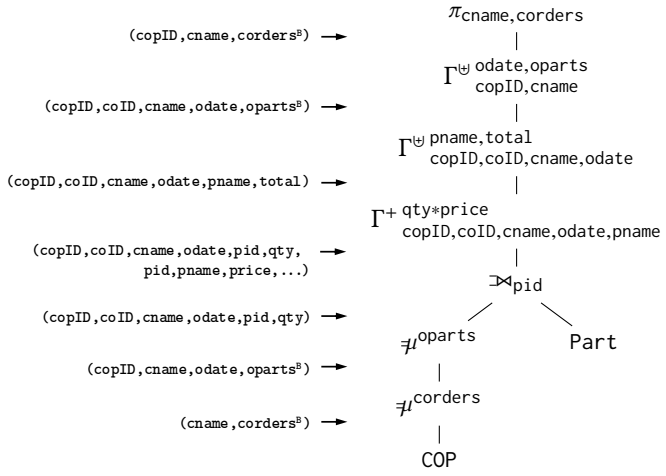


Figure 3: A query plan for the running example with output types. Bag types are annotated with ^{B}.

Example 2. Figure 3 shows a query plan output by an unnesting algorithm for the running example, along with the output type of each operator. For readability, we omit renaming operators from the query plan.

Starting at the bottom left, the plan performs an outer-unnest on *cop*.`corders`. To facilitate the upcoming nest operation at `corders`, the outer-unnest associates a unique ID to each tuple *cop*. The set $\mathcal{G}$ of grouping attributes at this point is {`copID`, `cname`}. The plan continues with an outer-unnest for *co*.`oparts`, extending each tuple *co* with a unique ID. The set of grouping attributes is {`copID`, `coID`, `cname`, `odate`}. Now the input is fully flattened. The next operation is an outer join between the plan built up to this point and the `Part` relation. This comprises the subplan corresponding to the input for the `sumBy` expression in the source query.

The `sumBy` expression is translated to a sum aggregate Γ^{+} 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 $\Gamma^{\uplus}$ created from entering the second level at *co*.`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 can optimize the plan from Figure 3 by projecting away unused attributes from the `Part` relation. We can push the sum aggregate Γ^{+} past the join to compute partial sums of `qty` values over the output of $\not\exists^{\mathtt{oparts}}$, 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

distributed at the granularity of top-level tuples. We translate operators of the plan language into operations over Datasets, taking into account that $\Gamma^{\uplus}$ and Γ^{+} 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 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 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 flat bag b^{F} of type T^{F} where each a_i has a special type 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 $T = \langle a_1 : T_1, \ldots, a_n : T_n \rangle$, T^{F} is formed by replacing each attribute a_i of bag type with a corresponding attribute of type Label. T^{D} includes attributes $a_i^{\mathtt{fun}}$ and $a_i^{\mathtt{child}}$ for each bag-valued a_i, where $a_i^{\mathtt{fun}}$ denotes the dictionary for a_i of type Label $\rightarrow$ Bag (T_i^{F}), while $a_i^{\mathtt{child}}$ denotes the dictionary tree of type T_i^{D}. Similarly, for a bag type $T = \mathsf{Bag}\,(T_1)$, we have $T^{\mathrm{F}} = \mathsf{Bag}\,(T_1^{\mathrm{F}})$ and $T^{\mathrm{D}} = T_1^{\mathrm{D}}$; for scalar types, we have $T^{\mathrm{F}} = T$ and $T^{\mathrm{D}} = \langle\rangle$.

Example 3. Based on the type of COP (Example 1), the shredded representation of COP consists of a top-level flat bag $\mathtt{COP}^{\mathrm{F}}$ of type Bag $(\langle\, \mathtt{cname} : \mathsf{string}\,,\, \mathtt{corders} : \mathsf{Label}\,\rangle)$ and a dictionary tree $\mathtt{COP}^{\mathrm{D}}$ of tuple type

$$\langle\, \mathtt{corders}^{\mathtt{fun}} : \mathsf{Label} \rightarrow \mathsf{Bag}\,(\langle\, \mathtt{odate} : \mathsf{date}, \mathtt{oparts} : \mathsf{Label}\,\rangle),$$
$$\mathtt{corders}^{\mathtt{child}} : \mathsf{Bag}\,(\langle\, \mathtt{oparts}^{\mathtt{fun}} : \mathsf{Label} \rightarrow$$
$$\mathsf{Bag}\,(\langle\, \mathtt{pid} : \mathsf{string}\,,\, \mathtt{qty} : \mathsf{real}\,\rangle), \mathtt{oparts}^{\mathtt{child}} : \mathsf{Bag}\,(\langle\,\rangle)\quad\rangle)\quad\rangle$$

The $\mathtt{COP}^{\mathrm{D}}$ dictionary tree encodes the $\mathtt{corders}^{\mathtt{fun}}$ dictionary for corders labels, the $\mathtt{oparts}^{\mathtt{fun}}$ dictionary for oparts labels, and the nesting structure via the child attributes. Since the type system prevents nesting tuples inside tuples, each child dictionary tree 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 $\mathrm{NRC}^{Lbl+\lambda}$, which extends NRC with a new atomic type for labels and a function type for dictionaries. The grammar of $\mathrm{NRC}^{Lbl+\lambda}$ is:

```
e ::= [Similar to Figure 1]
    | NewLabel(var, ...) | match e = NewLabel(var, ...) then e
    | Lookup(e, e) | MatLookup(e, e) | λvar .e
    | e DictTreeUnion e
T ::= [Similar to Figure 1]
    | Label                                  – Label Type
    | Label → Bag (F)                        – Dictionary Type
```

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

$$\mathtt{match}\ l = \mathtt{NewLabel}(\boldsymbol{x})\ \mathtt{then}\ F(\boldsymbol{x}, \boldsymbol{y})$$

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

We have the standard λ 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 Label $\rightarrow T$. We also have the standard function application: if e_1 is an expression of type Label $\rightarrow T$ and e_2 an expression of type Label, then we can form $\mathtt{Lookup}(\mathsf{e}_1, \mathsf{e}_2)$ of type

T. $\texttt{MatLookup}(\mathsf{e}_1, \mathsf{e}_2)$ corresponds to a lookup of the label returned by e_2 within the bag of label-bags pairs returned by e_1. Finally, we have a variation of the binary union for expressions representing dictionary trees, denoted as $\texttt{DictTreeUnion}$.

Query Shredding Algorithm. A 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.

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 $\mathcal{F}$ and $\mathcal{D}$; the full translation can be found in [39]. 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 base cases are those of constants and variables: for a scalar c, we have $\mathcal{F}(c) = c$ and $\mathcal{D}(c)$ is an empty dictionary tree (line 1); for a variable x of type T, the two functions return x^{F} and x^{D} whose types T^{F} and T^{D} depend on T as described above (line 2).

The interesting cases are those of tuple construction and tuple projection. For each bag-valued attribute a_i in a tuple constructor (line 3), $\mathcal{F}$ replaces the bag expression e_i with a new label encapsulating the (flat) free variables of e_i. The dictionary tree of the tuple constructor includes two attributes: a_i^{fun}, which represents the mapping from such a label to the flat variant e_i^{F} of e_i; and a_i^{child}, which represents the dictionary tree for e_i^{F}. To conform with the type system, each child dictionary tree is wrapped as a singleton bag as in Example 3. 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; thus, we omit it from the dictionary tree of the tuple constructor. When accessing a bag-valued attribute a in a tuple expression e (line 5), $\mathcal{F}$ returns a $\texttt{Lookup}$ construct computing the flat bag of $e.a$, based on the dictionary $\mathcal{D}(e).a^{\mathsf{fun}}$ 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 6), $\mathcal{F}$ recursively computes $e^{\mathrm{F}}.a$, while $\mathcal{D}$ returns an empty dictionary tree.

For the 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 $\texttt{DictTreeUnion}$ 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.

Example 4. We exhibit the shredding algorithm from Figure 4 on the query Q from Example 1. The algorithm produces a query Q^{F} computing the top-level flat bag and a query Q^{D} computing the dictionary tree for Q^{F}. We match Q to the $\texttt{for}$ construct in $\mathcal{F}$ (line 8) and recur to derive Q^{F}:

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

Recall that $\texttt{COP}^{\mathrm{F}}$ is a flat bag, so Q^{F} indeed computes a bag of flat tuples. We drop here the unused $\texttt{let}$ binding to cop^{D}. We derive Q^{D} after matching $\texttt{corders} := Q_{\texttt{corders}}$ in the top-level tuple constructor of Q (line 3):

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

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

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

□

Our implementation [40] refines the above shredding algorithm to retain only the relevant attributes of free variables in a $\texttt{NewLabel}$; for instance, the $\texttt{corders}$ labels in Q^{F} need to capture only $cop^{\mathrm{F}}.\texttt{corders}$ labels but not $cop^{\mathrm{F}}.\texttt{cname}$ 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 $Q^{\mathrm{D}}_{\mathsf{index}}$ in the dictionary tree, the transformation creates one assignment $\texttt{MatDict}_{\mathsf{index}} \Leftarrow e$, where the expression e computes a bag of tuples representing the materialized form of $Q^{\mathrm{D}}_{\mathsf{index}}$. 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 $\texttt{Lookup}$ on a symbolic dictionary by a $\texttt{MatLookup}$ on its materialized variant.

Materialization needs to resolve the domain of a symbolic dictionary. In our baseline materialization, the expression e computing $\texttt{MatDict}_{\mathsf{index}}$ 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 $Q^{\mathrm{D}}_{\mathsf{index}}$ for each label.

The materialization algorithm from Figure 5 produces a sequence of assignments given a shredded expression, its dictionary tree, and

	Pattern of e	$\mathcal{F}(e)$	$\mathcal{D}(e)$
1	c	c	$\langle\rangle$
2	var	var $^{\mathrm{F}}$	var $^{\mathrm{D}}$
3	$\langle \ldots a_i := e_i, \ldots$ e_i of bag type	$\langle \ldots a_i := \mathsf{NewLabel}((x^{\mathrm{F}})_{x \in \mathrm{F}\,\mathrm{V}}\ (e_i)), \ldots$	$\langle \ldots a_i^{\mathsf{fun}} := \lambda l.\ \mathsf{match}\ l = \mathsf{NewLabel}(\ldots)\ \mathsf{then}\ \mathcal{F}(e_i),$ $a_i^{\mathsf{child}} := \{\mathcal{D}(e_i)\}, \ldots$
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	$e.a$ of bag type	$\mathsf{Lookup}(\mathcal{D}(e).a^{\mathsf{fun}}, \mathcal{F}(e).a)$	$\mathsf{get}(\mathcal{D}(e).a^{\mathsf{child}})$
6	$e.a$ of scalar type	$\mathcal{F}(e).a$	$\langle\rangle$
7	$\{e\}$	$\{\mathcal{F}(e)\}$	$\mathcal{D}(e)$
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	$e_1 \uplus e_2$	$\mathcal{F}(e_1) \uplus \mathcal{F}(e_2)$	$\mathcal{D}(e_1)$ DictTreeUnion $\mathcal{D}(e_2)$
10	$op(e)$	$op(\mathcal{F}(e))$	$\mathcal{D}(e)$
11	$op(e_1, e_2)$	$op(\mathcal{F}(e_1), \mathcal{F}(e_2))$	$\langle\rangle$

Figure 4: Query shredding algorithm.

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, we first emit an assignment computing the set of labels produced in the parent assignment (lines 3-4). We then rewrite the dictionary a^{fun} 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 a^{fun} (lines 6-8), before recurring to the child dictionary tree (line 9).

EXAMPLE 5. We showcase the MATERIALIZE procedure on the shredded queries Q^{F} and Q^{D} from Example 4. Let MatCOP, $\mathsf{MatCOP}_{\mathsf{corders}}$, and $\mathsf{MatCOP}_{\mathsf{corders\_oparts}}$ denote the materializations of the top-level input bag $\mathsf{COP}^{\mathrm{F}}$ and the two symbolic dictionaries from $\mathsf{COP}^{\mathrm{D}}$, respectively. The procedure replaces COP^{F} by MatCOP in Q^{F} (line 1), followed by emitting the assignment to the variable TopBag:

```
TopBag ⇐ for cop^F in MatCOP union
           {⟨cname := cop^F.cname, corders := NewLabel(cop^F)⟩}
```

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

```
LabDomain_corders ⇐
  dedup(for x in TopBag union {⟨label := x.corders⟩})
```

The function uses the labels from $\mathsf{LabDomain}_{\mathsf{corders}}$ to materialize the $\mathsf{corders}^{\mathsf{fun}}$ dictionary (lines 6-8):

```
MatDict_corders ⇐
  for l in LabDomain_corders union
  {⟨label := l.label,
    value := match l.label = NewLabel(cop^F) then
      for co^F in MatLookup(MatCOP_corders, cop^F.corders)
      union {⟨odate := co^F.odate, oparts := NewLabel(co^F)⟩}
```

The query computing value corresponds to the body of the $\mathsf{corders}^{\mathsf{fun}}$ 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 $\mathsf{oparts}^{\mathsf{fun}}$ from $\mathsf{MatDict}_{\mathsf{corders}}$.

```
LabDomain_corders_oparts ⇐
 dedup(for x in MatDict_corders union {⟨label := x.oparts⟩})
MatDict_corders_oparts ⇐
  for l in LabDomain_corders_oparts union
   {⟨label :=l.label,
     value := match l.label = NewLabel(co^F) then
      sumBy^total_pname(
        for op^F in MatLookup(MatCOP_corders_oparts, co^F.oparts)
        union for p^F in Part^F union
                if (p^F.pid == op^F.pid) then
                  {⟨pname := p^F.pname,
                    total := op^F.qty * p^F.price⟩})⟩}
```

□

The materialization algorithm in Figure 5 omits the case of DictTreeUnion when traversing a dictionary tree to simplify presentation. The required extension is straightforward [39].

Domain Elimination. In many cases materialization produces label domains that are redundant. We optimize our materialization procedure using domain elimination rules.

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

```
λl. match l = NewLabel(x) then
      for y in Lookup(D, x.a) 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

MATERIALIZE(expression e^F, expression e^D, variable Top)

1 e_1^F = REPLACESYMBOLICDICTS(e^F) // by materialized dicts
2 EMIT(Top $\Leftarrow e_1^F$)
3 e_1^D = NORMALIZE(e^D) // recursively inline let bindings
4 MATERIALIZEDICT(e_1^D, Top)

MATERIALIZEDICT(expression e^D, variable $\text{Parent}_{\text{index}}$)

1 **let** $e^D = \langle a_1 := e_1, \ldots, a_n := e_n \rangle$
2 **foreach** $a^{\text{fun}} \in \{a_1, \ldots, a_n\}$
3 EMIT($\text{LabDomain}_{\text{index\_a}} \Leftarrow$
4 dedup(for x in $\text{Parent}_{\text{index}}$ union $\{ \langle \text{label} := x.a \rangle \}$))
5 fun = REPLACESYMBOLICDICTS($e^D.a^{\text{fun}}$)
6 EMIT($\text{MatDict}_{\text{index\_a}} \Leftarrow$
7 for l in $\text{LabDomain}_{\text{index\_a}}$ union
8 $\{ \langle \text{label} := l.\text{label}, \text{value} := fun(l.\text{label}) \rangle \}$)
9 MATERIALIZEDICT(get($e^D.a^{\text{child}}$), $\text{MatDict}_{\text{index\_a}}$)

Figure 5: Materialization algorithms.

the label-value pairs directly from the materialized dictionary MatD corresponding to D:

```
for z in MatD union
  { ⟨ label := z.label,
      value := let x := ⟨ a := z.label ⟩ in
               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 MatD.

EXAMPLE 6. Returning to Example 5, applying this rule avoids producing $\text{LabDomain}_{\text{corders}}$ as an intermediate result. The materialization of $\text{corders}^{\text{fun}}$ now corresponds to:

```
MatDict_corders ⇐
  for z in MatCOP_corders union
   { ⟨ label := z.label,
       value := for co^F in z.value union
                {⟨ odate := co^F.odate, oparts := NewLabel(co^F) ⟩} }
```

We can extend this rule to match a sumBy construct around the for construct, which also enables computing $\text{MatDict}_{\text{corders\_oparts}}$ directly from $\text{MatCOP}_{\text{corders\_oparts}}$. □

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

```
λl. match l = NewLabel(x) then
      for y in Y union if (y.a == x.b) then e
```

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

```
groupBy_label(for y in Y union
              let x := ⟨b := y.a⟩ in
              { ⟨ label := NewLabel(x), value := e ⟩ })
```

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 $\text{NRC}^{Lbl+\lambda}$, 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 with 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 cogroups.

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 1.

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 (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. 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 $\frac{100}{2.5} = 40$ 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. 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 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 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.

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

6 EXPERIMENTS

We evaluate the performance of our standard and shredded compilation routes and compare against existing systems that support nested data processing for both generated and real-world datasets. One goal of the experiments is to evaluate how the succinct dictionary representation of shredding compares to the flattening methods. We look at how the strategies scale with increased levels of nesting, number of top-level tuples, and the size of inner collections. We explore how aggregations can reduce dictionaries and downstream dictionary operations for nested and flat output types. Finally, to highlight skew-handling, we evaluate the performance of our framework for increasing amounts of skew.

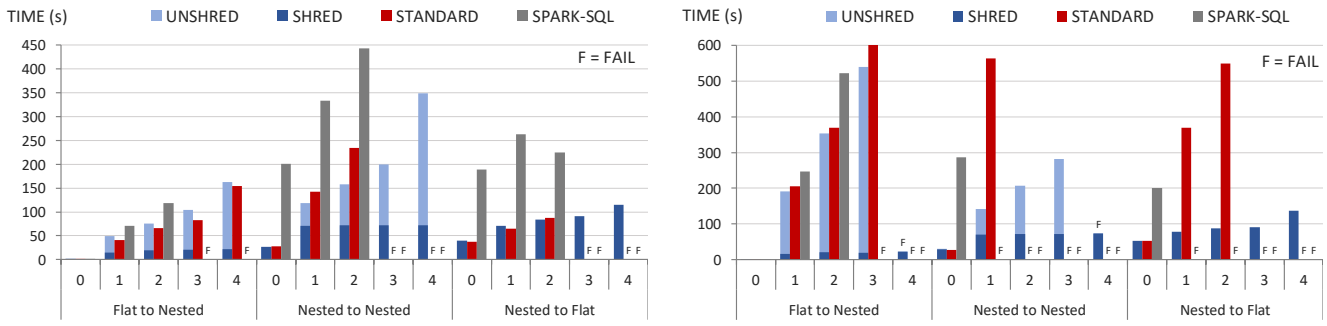
Our experimental results can be summarized as follows:

- 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 micro-benchmark 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



(a) Narrow schema

(b) Wide schema

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

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 $\mathsf{sumBy}^{\mathtt{qty*price}}_{\mathsf{pname}}$, as in Example 1. The nested-to-nested queries thus produce the same hierarchy as the flat-to-nested queries. Finally, the nested-to-flat queries follow the same construction as the nested-to-nested queries, but apply $\mathsf{sumBy}^{\mathtt{qty*price}}_{name}$ 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, 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 generator.

The biomedical benchmark includes an end-to-end pipeline E2E, based on [47], consisting of 5 steps. The inputs include a two-level nested relation BN_2 (280GB) [26, 30], a one-level nested relation BN_1 (4GB) [42], and five relational inputs – the most notable of which are BF_1 (23G), BF_2 (34GB), and BF_3 (5KB) [18, 26]. The first two steps of E2E are the most expensive. $\textsc{Step}_1$ flattens the whole of BN_2, while performing a nested join on each level (BF_2 at level 1 and BF_3 at level 2), aggregating the result, and finally grouping to produce nested output. $\textsc{Step}_2$ joins and aggregates BN_1 on the first-level of the output of $\textsc{Step}_1$.

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]. 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. Standard also serves as a means to explore the general functionality of flattening methods. 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 $\textsc{Standard}_{\textsc{skew}}$, $\textsc{Shred}_{\textsc{skew}}$, and $\textsc{Shred}^{+\textsc{U}}_{\textsc{skew}}$ 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.

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 $\texttt{sumBy}^{\texttt{qty*price}}_{\texttt{pname}}$, 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

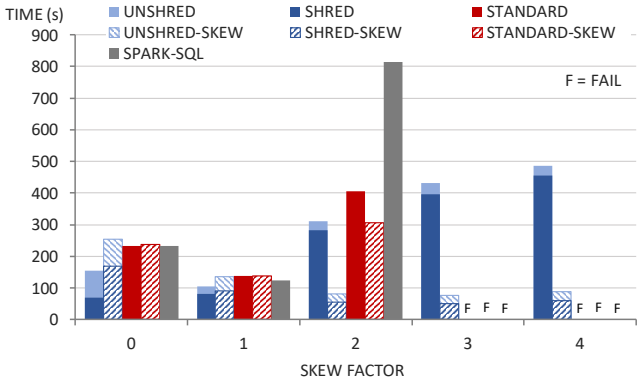


Figure 8: Nested-to-nested narrow TPC-H query with two levels of nesting on increasingly skewed datasets.

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-unaware 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. $\text{Shred}_{\text{skew}}$ has up to a 15x performance gain over other methods for moderate amounts of skew. The skew join of $\text{Shred}_{\text{skew}}$ 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, Std, and $\text{Standard}_{\text{skew}}$ 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

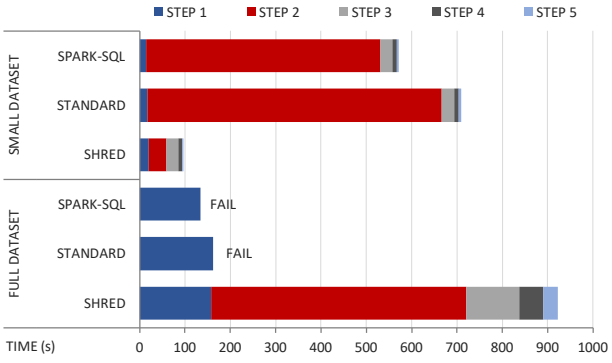


Figure 9: End-to-end pipeline of the biomedical benchmark.

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 SparkSQL, 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 Step_2, we also provide results with a smaller dataset using only 6GB BN_2, 2GB BF_1, and 4GB BF_2. Shred exhibits many advantages, overall surviving the whole pipeline. The results for Step_1 using the small and full dataset exhibit similar behavior to the nested-to-flat results for TPC-H. Since the whole BN_2 input is flattned and there are many projections during Step_1, Standard and SparkSQL are not too burdened by the nested joins that occur while flattening.

The methods diverge at Step_2, where a nested join between BN_1 and the output of Step_1 leads to an explosion in the amount of data being shuffled. Shred displays up to a 16x performance gain for Step_2 with the small dataset. Despite the small number of output attributes, the result of the join between flattened output of Step_1 and BN_1 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 Step_1, 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, DIQL [20] and MRQL [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

[1] Foto N Afrati, Dan Delorey, Mosha Pasumansky, and Jeffrey D Ullman. 2014. Storing and querying tree-structured records in Dremel. PVLDB 7, 12 (2014), 1131–1142.

[2] Alexander Alexandrov, Asterios Katsifodimos, Georgi Krastev, and Volker Markl. 2016. Implicit Parallelism through Deep Language Embedding. SIGMOD Rec. 45, 1 (2016), 51—-58.

[3] Apache Cassandra. 2020. (2020). cassandra.apache.org.

[4] Apache CouchDB. 2020. (2020). couchdb.apache.org.

[5] Apache Hive. 2020. (2020). hive.apache.org.

[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In SIGMOD . 1383–1394.

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

[8] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. 2010. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In Cloud Computing .

[9] Paul Beame, Paraschos Koutris, and Dan Suciu. 2017. Communication Steps for Parallel Query Processing. J. ACM 64, 6 (2017), 40:1–40:58.

[10] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. Theoretical Computer Science 149, 1 (1995), 3–48.

[11] Jesús Camacho-Rodríguez, Dario Colazzo, Ioana Manolescu, and Juan A.M. Naranjo. 2015. PAXQuery: Parallel Analytical XML Processing. In SIGMOD .

[12] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. PVLDB 1, 2 (2008), 1265–1276.

[13] James Cheney, Sam Lindley, and Philip Wadler. 2014. Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets. In SIGMOD .

[14] Citus. 2020. (2020). www.citusdata.com.

[15] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In Formal Methods for Components and Objects .

[16] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 1 (2008), 107–113.

[17] Jan Van den Bussche. 2001. Simulation of the Nested Relational Algebra by the Flat Relational Algebra. Theor. Comput. Sci. 254, 1-2 (2001), 363–377.

[18] Karen Eilbeck, Suzanna E Lewis, Christopher J Mungall, Mark Yandell, Lincoln Stein, Richard Durbin, and Michael Ashburner. 2005. The Sequence Ontology: A tool for the unification of genome annotations. Nature Methods 6 (2005), R44.

[19] Leonidas Fegaras. 2017. An algebra for distributed Big Data analytics. Journal of Functional Programming 27 (2017).

[20] Leonidas Fegaras. 2019. Compile-Time Query Optimization for Big Data Analytics. Open Journal of Big Data (OJBD) 5, 1 (2019), 35–61.

[21] Leonidas Fegaras and David Maier. 2000. Optimizing Object Queries Using an Effective Calculus. TODS 25, 4 (2000), 457–516.

[22] Leonidas Fegaras and Md Hasanuzzaman Noor. 2018. Compile-Time Code Generation for Embedded Data-Intensive Query Languages. In BigData Congress .

[23] Stella Giannakopoulou, Manos Karpathiotakis, Benjamin Gaidioz, and Anastasia Ailamaki. 2017. CleanM: An Optimizable Query Language for Unified Scale-out Data Cleaning. PVLDB 10, 11 (2017), 1466—-1477.

[24] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. 2009. FERRY: Database-Supported Program Execution. In SIGMOD .

[25] Torsten Grust, Jan Rittinger, and Tom Schreiber. 2010. Avalanche-Safe LINQ Compilation. PVLDB 3, 1–2 (2010), 162—-172.

[26] ICGC 2020. International Cancer Genome Consortium. (2020). https://icgc.org/.

[27] Won Kim. 1982. On Optimizing an SQL-like Nested Query. TODS 7, 3 (1982), 443—-469.

[28] Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In PODS.

[29] Paraschos Koutris, Semih Salihoglu, and Dan Suciu. 2018. Algorithmic Aspects of Parallel Data Processing . Now Publishers Inc.

[30] William McLaren, Laurent Gil, Sarah E. Hunt, Harpreet Singh Riat, Graham R. S. Ritchie, Anja Thormann, Paul Flicek, and Fiona Cunningham. 2016. The Ensembl Variant Effect Predictor. Genome Biology 17, 1 (2016), 122.

[31] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In SIGMOD .

[32] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. PVLDB 3, 1-2 (2010), 330–339.

[33] MongoDB. 2020. (2020). www.mongodb.com.

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

[35] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. 2016. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In ICDE .

[36] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, and others. 2018. F1 Query: Declarative Querying at Scale. PVLDB 11, 12 (2018), 1835–1848.

[37] Kazunori Sato. 2012. An Inside Look at Google BigQuery. White Paper, Google Inc (2012), 12.

[38] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, and et al. 2013. F1: A Distributed SQL Database That Scales. PVLDB 6, 11 (2013), 1068—-1079.

[39] Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. 2020. Scalable Querying of Nested Data. (2020). arxiv.org/abs/2011.06381.

[40] Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. 2020. Scalable Querying of Nested Data. (2020). github.com/jacmarjorie/trance.

[41] Dan Suciu. 1996. Implementation and Analysis of a Parallel Collection Query Language. In VLDB .

[42] D Szklarczyk, AL Gable, and D et al. Lyon. 2019. STRING v11: protein-protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. Nucleic Acids Res 47, D1 (2019), D607–D613.

[43] TCPH 2020. Transaction Processing Performance Council. (2020). TPC-H Benchmark . http://www.tpc.org.

[44] Limsoon Wong. 1994. Querying Nested Collections . Ph.D. Dissertation. Univ. Pennsylvania.

[45] Limsoon Wong. 2000. Kleisli, a functional query system. Journal of Functional Programming 10, 1 (2000), 19–56.

[46] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, and others. 2010. Spark: Cluster Computing with Working Sets. HotCloud 10, 10-10 (2010), 95.

[47] Wei Zhang and Shu-Lin Wang. 2020. A Novel Method for Identifying the Potential Cancer Driver Genes Based on Molecular Data Integration. Biochemical Genetics 58, 1 (2020), 16–39.