Supplementary material for Scalable Querying of Nested Data

1 Nested TPC-H Benchmark

We detail the nested TPC-H benchmark introduced in our experimental evaluation (Section 7). The queries are designed for a systematic exploration of nested queries within a distributed environment, focusing on a small number of top-level tuples and large inner collections. The queries range from 0 to 4 levels of nesting, organized such that the number of top-level tuples decrease as the level of nesting increases.

For each of the query categories below, we provided the input NRC, the optimal plan produced from the standard pipeline (FLAT) and the shredded pipeline (SHRED). Where relevant, we describe the plan for unshredding and discuss optimizations introduced by the code generator.

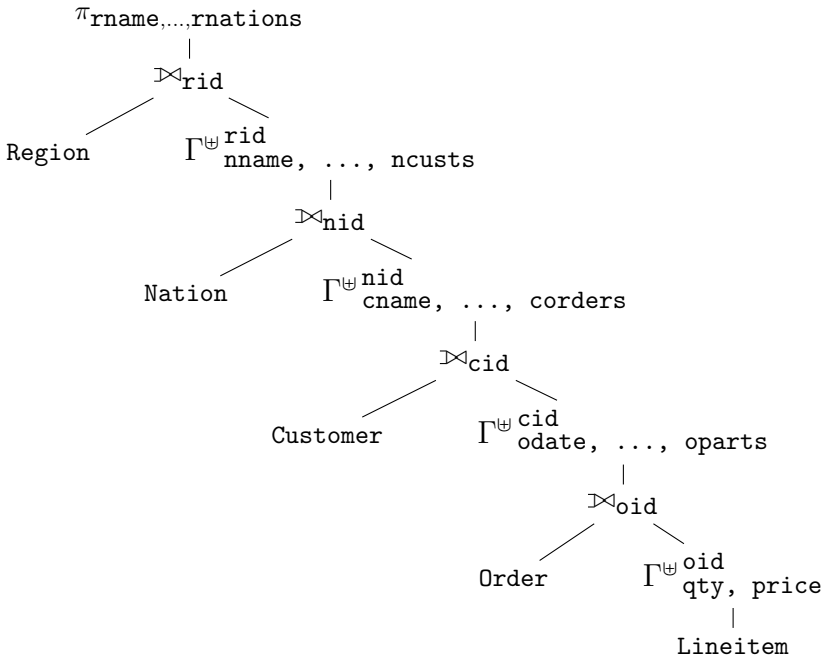
1.1 Flat-to-nested

Here we detail the flat-to-nested queries of the benchmark, which build up nested objects from flat input. The flat-to-nested queries include the Lineitem table (0 levels), Lineitem grouped by Orders (`oparts`), `oparts` grouped by Customers (`corders`), `corders` grouped by Nation (`ncusts`), `ncusts` grouped by Region (`rnations`). For scale factor 100, this organization produces queries with 600 million, 150 million, 15 million, 25, and 5 top level tuples. These queries are implemented with and without projection. The ellipses represent the additional fields that may be present based on projections. We provide the query with 4 levels, since queries with different levels are merely subsets of this query.

1.1.1 Input NRC

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

1.1.2 Plan produced by the standard pipeline



The sequential join-nest operations in the above plan will be merged into cogroups during code generation (Section 4.2). We implement the cogroup in a left-outer fashion, persisting empty bags from the right relation for every matching tuple in the left relation. As an example, consider the join and nest over `Order` and `Lineitem` in the above.

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

1.1.3 Plan produced by the shredded pipeline

The below is the plan produced by the shredded pipeline for the evaluation of the shredded query prior to unshredding (reconstructing the nested object). The plan produced by the shredded pipeline for unshredding is identical to the plan produced by the standard pipeline, with each input relation represented as a top-level bag.

$$\texttt{RNCOP}_{Top} := \pi_{\texttt{rname},\ldots,\texttt{rnations}:=\texttt{rid}}(\texttt{Region})$$

$$\texttt{rnations}_{Dict} := \pi_{\texttt{label}:=\texttt{rid},\texttt{nname},\ldots,\texttt{ncusts}:=\texttt{nid}}(\texttt{Nation})$$

$$\texttt{ncusts}_{Dict} := \pi_{\texttt{label}:=\texttt{nid},\texttt{cname},\ldots,\texttt{corders}:=\texttt{cid}}(\texttt{Customer})$$

$$\texttt{corders}_{Dict} := \pi_{\texttt{label}:=\texttt{cid},\texttt{odate},\ldots,\texttt{oparts}:=\texttt{oid}}(\texttt{Order})$$

$$\texttt{oparts}_{Dict} := \pi_{\texttt{label}:=\texttt{oid},\texttt{pid},\texttt{qty}}(\texttt{Lineitem})$$

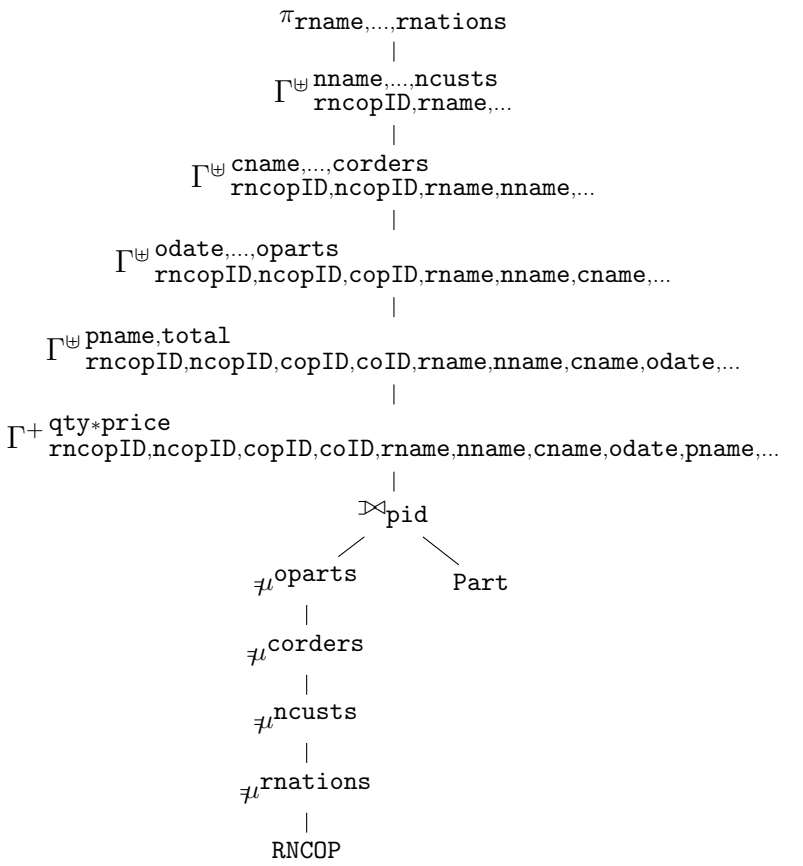
1.2 Nested-to-nested

The nested-to-nested queries take the materialized result of the flat-to-nested queries as input and perform nested operations at the lowest level. The levels continue reflecting the same hierarchy and number of top-level tuples as the flat-to-nested queries. These are queries that operate on nested input. The ellipses represent the additional fields that may be present in queries with a varying number of projections. We provide the query for 4 levels of nesting, all other queries can be derived from this query.

1.2.1 Input NRC

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

1.2.2 Plan produced by the standard pipeline



1.2.3 Plan produced by the shredded pipeline

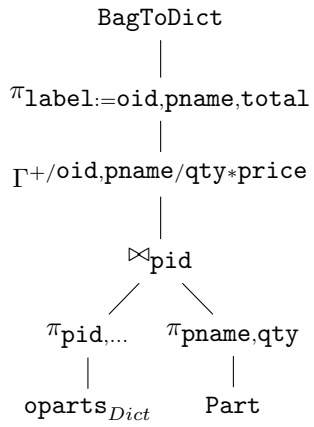
$\texttt{RNCOP}_{Top} := \pi_{\texttt{rname},\ldots,\texttt{rnations}:=\texttt{rid}}(\texttt{RNCOP}_{Top})$

$\texttt{rnations}_{Dict} := \pi_{\texttt{label}:=\texttt{rid},\texttt{nname},\ldots,\texttt{ncusts}:=\texttt{nid}}(\texttt{rnations}_{Dict})$

$\texttt{ncusts}_{Dict} := \pi_{\texttt{label}:=\texttt{nid},\texttt{cname},\ldots,\texttt{corders}:=\texttt{cid}}(\texttt{ncusts}_{Dict})$

$\texttt{corders}_{Dict} := \pi_{\texttt{label}:=\texttt{cid},\texttt{odate},\ldots,\texttt{oparts}:=\texttt{oid}}(\texttt{corders}_{Dict})$

$\texttt{oparts}_{Dict} :=$



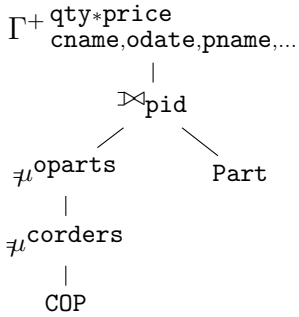
1.3 Nested-to-flat

The nested-to-flat queries take the materialized result of the flat-to-nested queries as input, apply an aggregation at the top-level, and thus work on the fully flattened input. We show the query for two levels of nesting, since this is the query described in the flat-to-nested experiments (Section 7).

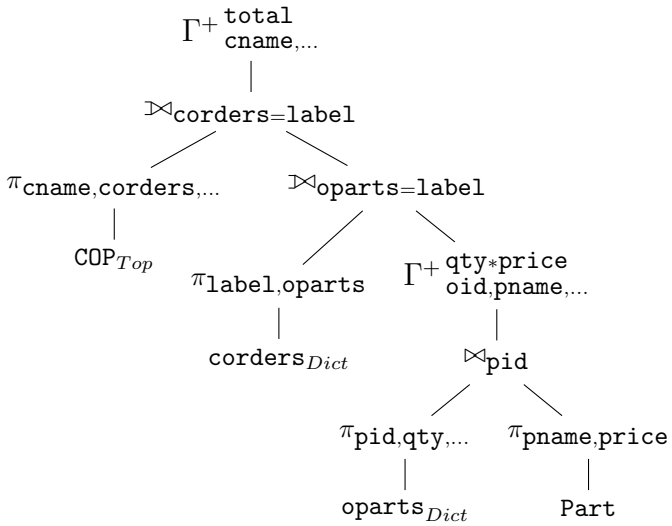
1.3.1 Input NRC

```
sumBy^total_cname,...(
  for c in COP union
    for o in c.corders union
      for l in o.oparts union
        for p in Part union
          if l.pid == p.pid then
            {⟨ cname := c.cname, ...
                total := l.qty * p.price ⟩})
```

1.3.2 Plan produced by the standard pipeline



1.3.3 Plan produced by the shredded pipeline



2 Spark Implementation of the Plan Operators

Figure 1 details the Spark implementations of the plan operators (Section 2.2) applied during code generation (Section 4.2). All operations use Dataset[R], where `R` is an arbitrary case class. For $\not\mu^a\ (X)$, we use `index` to create a unique index associated to the top-level input of the unnest operator.

As mentioned in Section 4.2, Spark Datasets are a special instance of RDD that use encoders to avoid the large memory footprint of using RDDs of case classes. Though similar in their underlying data type, the Dataset and RDD APIs are different and thus will have different implementations for the language operators. Figure 2 provides the implementation of the operators of the plan language using Spark RDDs. In the next section, we will highlight the advantages of using Spark Datasets as the underlying type in the code generator.

Plan Operator	Definition for Dataset API
$\sigma_{p(x)}(X)$	`X.filter(x => p(x))`
$\pi_{a_1,\ldots,a_k}(X)$	`X.select(a1, ..., ak)`

$\mu^{a_i}(X)$	`// R does not contain x.ai`
	`X.flatMap(x => x.ai.map(y =>`
	`  R(x.a1, ..., x.ak, y.b1, ..., y.bj))).as[R]`

$\dot{\mu}^{a}(X)$	`// R does not contain x.ai`
	`X.withColumn(index, monotonically_increasing_id()).flatMap(x =>`
	`  if (x.ai.isEmpty)`
	`  R(x.index, x.a1, ..., x.ak, None, ..., None)`
	`  else x.ai.map(y =>`
	`  R(x.index, x.a1, ..., x.ak, Some(y.b1), ...,`
	`  Some(y.bj)))).as[R]`

$X \bowtie_{f(x)=g(y)} Y$	`X.join(Y, f === g)`
$X \rtimes_{f(x)=g(y)} Y$	`X.join(Y, f === g, "left_outer")`
$\Gamma^{value(x)}_{+,key(x)}(X)$	`X.groupByKey(x => key(x)).agg(typed.sum(x =>` `  value(x) match { case Some(v) => v; case _ => 0 }))`

$\Gamma^{value(x)}_{\uplus,key(x)}(X)$	`X.groupByKey(x => key(x)).mapGroups{ case (key, values) =>`
	`  val grp = values.flatMap{ case x =>`
	`  value(x) match { case Some(t) => Seq(t); case _ => Seq()`
	`  }}.toSeq`
	`  (key, grp)`
	`}`

Figure 1: Plan language operators and their semantics for Spark Datasets.

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

$\not\mu^{a}(X)$	`// x.dropAddIndex(a) excludes attribute a from tuple x,`
	`// and adds a unique index for every x-tuple`
	`X.flatMap(x => if (x.a.isEmpty) Vector((x, null))`
	`  else x.a.map(y => (x.dropAddIndex(a), y)))`

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

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

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

Figure 2: Plan language operators and their semantics for Spark RDDs.

3 Shredding: Union Materialization Case

In Section 5.4, we describe the process of creating domains for homogeneous labels during materialization. Here, we detail the transformation that splits domains in the case where the transformation is not guaranteed to return tuples of the same type (ie. union).

The basic transformation is of the form $\mathcal{M}[\![\langle E_\sigma : \sigma_0 \preceq \sigma\rangle]\!]_{\tau,\sigma_0}$, where σ_0 is a dictionary index, each E_σ is an expression returning a dictionary, with the indices σ ranging over all valid indices extending index σ_0, and τ is a tag that can be associated with a label returned in one of the dictionaries E_σ. This transformation proceeds by induction on the dictionary index of the output type.

The inductive transformation first outputs the statement:

$\textsc{MatDict}_{\sigma,\tau} \Leftarrow$
  for l in $\textsc{LabDomain}_{\sigma,\tau}$ union
    $\{(\langle \texttt{label} := l.\texttt{label}, \texttt{value} := \texttt{Lookup}(E^{\texttt{DICT}}_{\sigma_0}, l)\rangle\}$

Then for each attribute a_i such that $\sigma_0\ a_i$ is a valid dictionary index of T and each tag τ' that can be associated with one of the dictionaries E_σ with σ extending $\sigma_0\ a_i$, this transformation outputs the statement:

$\textsc{LabDomain}_{\sigma_0\ a_i,\tau'} \Leftarrow$
  for t in $\textsc{MatDict}_{\sigma,\tau}$ union
    if $(\textsc{Extrcttag}(t) == \tau)$ then $\{\langle \texttt{label} := \ t.\texttt{label}\rangle\}$

and the statement:

$\mathcal{M}[\![\langle E_\sigma \sigma_0\ \ a_i \preceq \sigma\rangle]\!]_{\tau'} \sigma_0 a_i$

where $\sigma_0\ a_i$ ranges over all indices extending σ_0. Extrcttag is a function that extracts the tag from the tuple in question.

The top-level call to materialize will take an additional expression E^{FLAT}, and will return $V^{\text{FLAT}} \Leftarrow E^{\text{FLAT}}$ followed by the label domain materialization statements and recursive calls as above. This process splits label domains for each encoded type ensuring that label domains are homogeneous in the presence of union operators.

4 Additional Experimental Results

This section contains additional experimental results. All results are for 100GB of non-skewed TPC-H data (scale factor 100, skew factor 0). We use the flat-to-nested and nested-to-nested queries to compare the performance of the standard and shredded pipeline using both RDDs and Datasets. We also explore the effects of introducing database-style optimizations on the plans of the standard pipeline. The results highlight advantages of using Datasets over RDDs in code generation, particularly for nested collections. The results also show how introducing database-style optimizations can significantly improve performance of the standard pipeline, generating programs similar to programs that have been optimized by hand.

4.1 Performance comparison of Spark RDDs and Datasets

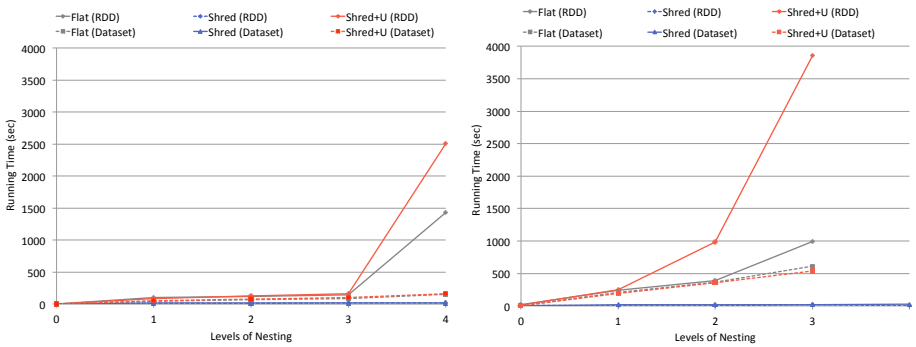
We use the flat-to-nested and nested-to-nested queries from the benchmark to compare the performance of FLAT (standard pipeline), SHRED (shredded pipeline without unshredding), and SHRED^{+U} (shredded pipeline with unshredding) using RDDs and Datasets. Figure 3 displays the results for the flat-to-nested queries with (3a) and without projections (3b). With projections, the results show that all strategies exhibit similar performance up to three levels of nesting. The strategies diverge at four levels of nesting where SHRED^{+U} and FLAT with RDDs have a spike in total run time. The results without projections follow a similar trend, with strategies diverging earlier at lower levels of nesting. Without projections, SHRED^{+U} with RDDs shows increasingly worse performance starting at two levels of nesting. SHRED with RDDs and SHRED with Datasets have similar performance.

As stated in the flat-to-nested benchmark section, the plan produced for unshredding in the shredded pipeline and the plan for the standard pipeline are identical; thus, the difference in performance is in code generation for the unshredding procedure. Both methods use a series of cogroups to build up a nested set; however, unshredding requires additional map operations and intermediate object creation that are required for reconstructing nested objects from dictionaries. Case classes lacking binary encoders require a significant amount of time and space to create and store, which only gets more expensive with increasing levels of nesting. SHRED^{+U} with RDDs grows exponentially as the number of nesting increases, bringing along the previous level with each level of nesting. This is also a problem for FLAT, which sees worse performance with increasing levels of nesting but grows at a slower rate due to less object creation.

To consider what this means from a code generation perspective, compare the project operator of Figure 1 to the project operator in Figure 2. The RDD API maps over a relation and creates a new case class (`R`), whereas the Dataset API avoids this map and uses a select operator that accepts a series of attribute names. By explicitly stating the attributes, the Dataset API has delayed an explicit map operation allowing for further performance benefits from the Spark optimizer. Further, when the time comes to construct `R` objects, the Dataset API will leverage the binary format and incur a much smaller memory footprint.

Figure 4 further highlights the benefits of Datasets with nested-to-nested queries. Both with and without projections, the difference between SHRED with RDDs and SHRED with Datasets shows a 2× performance improvement of the explicit statement of attributes within the Dataset API. In this case, we also see that FLAT has decreased performance with RDDs. While the unnest operators used in the code generators both use flat-map operations, the Dataset API maintains a low-memory footprint for the newly created objects (`.as[R]` in the code generator).

These results show that code generation with Datasets has minimized overhead in object creation and gains further improvements from passing meta-information to the Spark optimizer. Beyond the application to Spark, these



(a) With projections

(b) Without projections (wide)

Figure 3: Performance comparison of RDDs and Datasets for the flat-to-nested benchmarked queries.

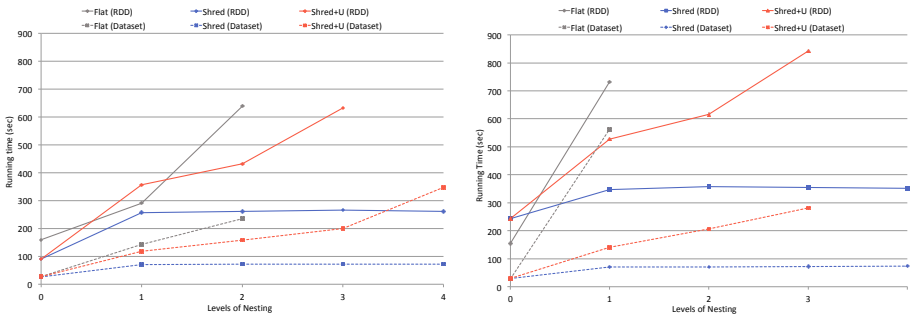
results should be useful for further implementations of automated nested query processing on distributed systems.

4.2 Standard pipeline optimizations

This experiment highlights how the standard pipeline can leverage database-style optimizations to automatically generate programs that are comparable to hand-optimized programs. We use the flat-to-nested and nested-to-nested queries for plans from the standard pipeline that have an increasing amount of optimizations applied. Figure 5 shows the results of this experiment. Flat with no optimizations is the untouched plan that comes out of the unnesting algorithm. Flat with pushed projections is the plan from the unnesting algorithm with projections pushed. Flat is the standard pipeline used in all experiments with projections pushed, nests pushed past joins, and merged into cogroups where relevant. Flat is comparable to a highly optimized, manually defined program over nested collections.

The results show that even simple optimizations like pushing projections can provide major performance benefits for flattening methods. For example, Figure 5a shows that projections have not only increased performance of the standard pipeline, but have allowed the strategy to survive to deeper levels of nesting. This is expected since we have already seen that Flat performance is heavily impacted by the presence of projections (ie. the number of attributes in the output tuples). For nested-to-nested queries, Flat is the only strategy to survive past one level of nesting. These results show that database-style optimizations are not only beneficial to improve performance, but are necessary when using flattening methods with only shallow-nested objects.

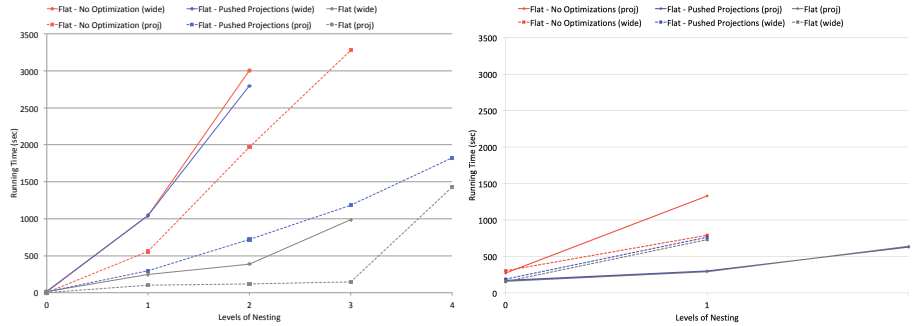
The current state-of-the-art for defining programs over distributed, nested collections is hand-written code that has been manually optimized by an ex-



(a) With projections

(b) Without projections (wide)

Figure 4: Performance comparison of RDDs verse Datasets for the nested-to-nested benchmarked queries.



(a) Flat to Nested

(b) Nested to Nested

Figure 5: Performance comparison of benchmarked queries for increasing optimization levels of the standard pipeline.

perienced developer. This is part of the programming mismatch we describe in the Introduction (Section 1). These results show that leveraging standard database-style optimizations can be useful in the automation of programs that operate on nested collections, and supports more systematic benchmarking of flattening methods than manually optimizing hand-written code.