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. All queries start with the Lineitem table at level 0, then group across Orders, Customer, Nation, then Region, as the level increases. Each query has a *wide* variant where we keep all the attributes, and a *narrow* variant which follows the grouping with a projection at each level.

For each of the query categories below, we provided the input NRC, the optimal plan produced from the standard pipeline (STD) 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. 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. This starts with the Lineitem table (0 levels), Lineitem grouped by Orders (`oparts`), `oparts` grouped by Customers (`corders`), `corders` grouped by Nation (`ncusts`), `ncusts` grouped by Region (`rnations`). At the lowest level we keep the partkey and quantity of a Lineitem. At the higher levels the thin variant keeps only a single attribute, e.g. orderdate for Orders, customername for Customer, etc.

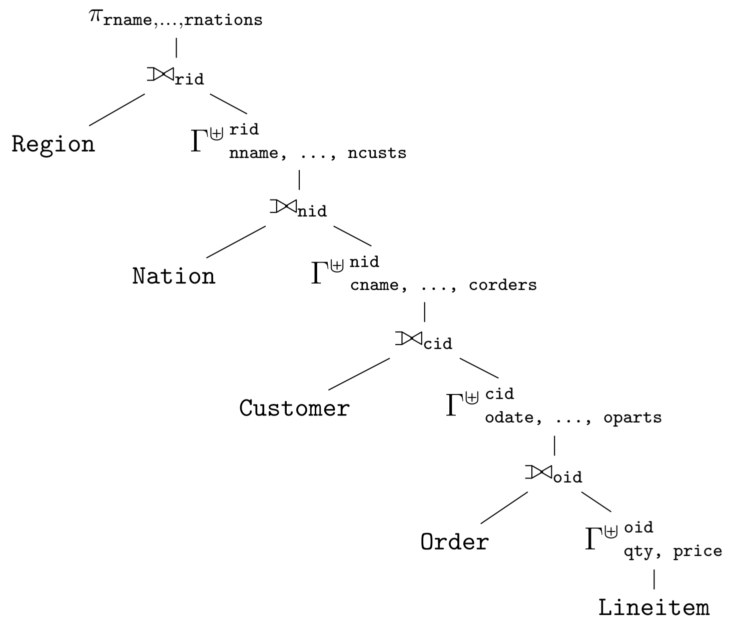
The ellipses represent the additional fields that may be present based on narrow and wide versions. We provide the query with 4 levels, since queries with different levels are merely subsets of this query.

1.1.1 Input NRC

NRC Program.

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

1.1.2 Plan produced by the standard pipeline


$\pi_{\text{rname},\ldots,\text{rnations}}$
$\bowtie_{\text{rid}}$
Region
$\Gamma^{\uplus}{}^{\text{rid}}_{\text{nname, ..., ncusts}}$
$\bowtie_{\text{nid}}$
Nation
$\Gamma^{\uplus}{}^{\text{nid}}_{\text{cname, ..., corders}}$
$\bowtie_{\text{cid}}$
Customer
$\Gamma^{\uplus}{}^{\text{cid}}_{\text{odate, ..., oparts}}$
$\bowtie_{\text{oid}}$
Order
$\Gamma^{\uplus}{}^{\text{oid}}_{\text{qty, price}}$
Lineitem

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.

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

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

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

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

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

1.2 Nested-to-nested

The *nested-to-nested* queries take the materialized result of the flat-to-nested queries as input and perform a join with `Part` at the lowest level, followed by $\texttt{sumBy}^{\texttt{qty}\times\texttt{price}}_{\texttt{pname}}$, as in Example 1. The nested-to-nested queries thus produce the same hierarchy as the flat-to-nested queries. 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 the narrow and wide version of the queries. 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 :=
            for o in c.corders union
              {⟨odate := o.odate, ..., oparts :=
                sumBy^{total}_{pname}(
                  for l in o.oparts union
                    for p in Part union
                      if l.pid == p.pid then
                        {⟨pname := p.pname,
                          total := l.qty * p.price⟩})⟩}⟩}⟩}⟩}
```

1.2.2 Plan produced by the standard pipeline

$$\begin{tikzcd}
 & \pi_{\texttt{rname},\ldots,\texttt{rnations}} \arrow[d, dash] & \\
 & \Gamma^{\uplus\,\texttt{nname},\ldots,\texttt{ncusts}}_{\texttt{rncopID},\texttt{rname},\ldots} \arrow[d, dash] & \\
 & \Gamma^{\uplus\,\texttt{cname},\ldots,\texttt{corders}}_{\texttt{rncopID},\texttt{ncopID},\texttt{rname},\texttt{nname},\ldots} \arrow[d, dash] & \\
 & \Gamma^{\uplus\,\texttt{odate},\ldots,\texttt{oparts}}_{\texttt{rncopID},\texttt{ncopID},\texttt{copID},\texttt{rname},\texttt{nname},\texttt{cname},\ldots} \arrow[d, dash] & \\
 & \Gamma^{\uplus\,\texttt{pname},\texttt{total}}_{\texttt{rncopID},\texttt{ncopID},\texttt{copID},\texttt{coID},\texttt{rname},\texttt{nname},\texttt{cname},\texttt{odate},\ldots} \arrow[d, dash] & \\
 & \Gamma^{+\,\texttt{qty*price}}_{\texttt{rncopID},\texttt{ncopID},\texttt{copID},\texttt{coID},\texttt{rname},\texttt{nname},\texttt{cname},\texttt{odate},\texttt{pname},\ldots} \arrow[d, dash] & \\
 & \leftouterjoin_{\texttt{pid}} \arrow[dl, dash] \arrow[dr, dash] & \\
\mu^{\texttt{oparts}} \arrow[d, dash] & & \texttt{Part} \\
\mu^{\texttt{corders}} \arrow[d, dash] & & \\
\mu^{\texttt{ncusts}} \arrow[d, dash] & & \\
\mu^{\texttt{rnations}} \arrow[d, dash] & & \\
\texttt{RNCOP} & &
\end{tikzcd}$$

1.2.3 Plan produced by the shredded pipeline

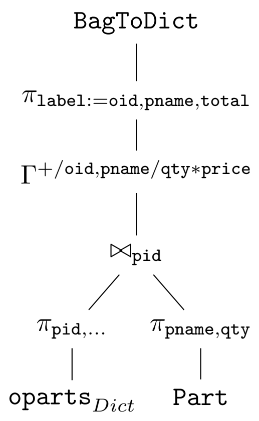
$\texttt{RNCOP}_{Top} :=\ \pi_{\texttt{rname},\ldots,\texttt{rnations}:=\texttt{rid}}(\texttt{RNCOP}_{Top})$

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

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

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

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


BagToDict
$\pi_{\texttt{label:=oid,pname,total}}$
$\Gamma^{+/\texttt{oid,pname}/\texttt{qty*price}}$
$\bowtie_{\texttt{pid}}$
$\pi_{\texttt{pid},\ldots}$
$\pi_{\texttt{pname,qty}}$
$\texttt{oparts}_{Dict}$
Part

1.3 Nested-to-flat

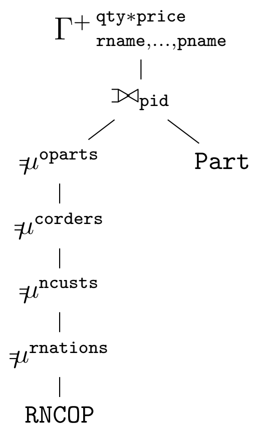
The *nested-to-flat* queries follow the same construction as the nested-to-nested queries, but apply $\texttt{sumBy}^{\texttt{qty}\times\texttt{price}}_{name}$ at top-level, where *name* is one of the top-level attributes; this returns a flat collection persisting only attributes from the outermost level.

The ellipses represent the additional fields that may be present in the narrow and wide version of the queries. We provide the query for 4 levels of nesting, all other queries can be derived from this query.

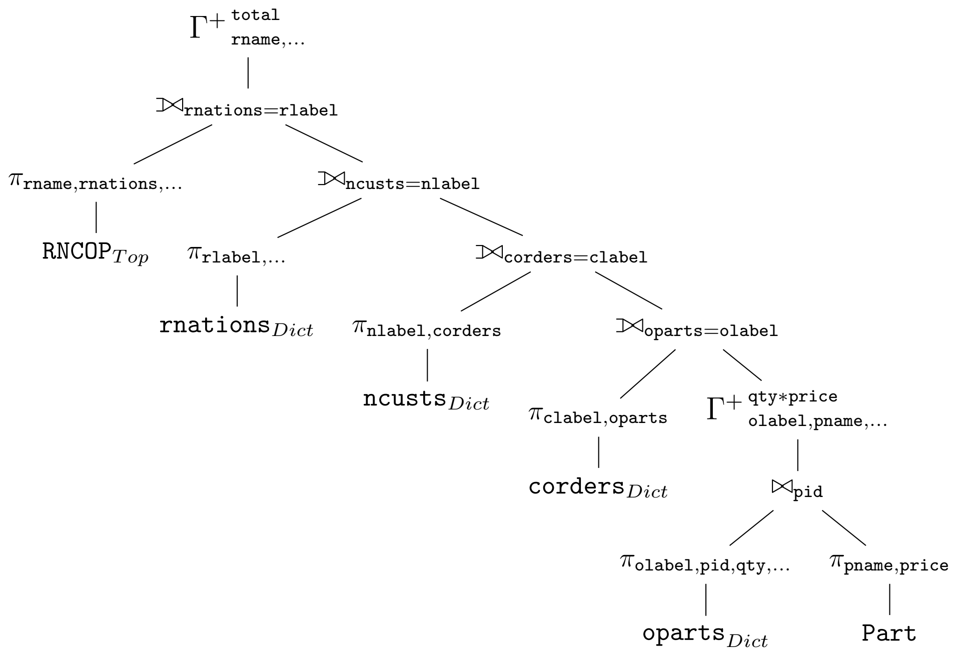
1.3.1 Input NRC

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

1.3.2 Plan produced by the standard pipeline



Plan produced by the shredded pipeline.


$\Gamma^{+\,\texttt{total}}_{\texttt{rname},\ldots}$
$⟕_{\texttt{rnations=rlabel}}$
$\pi_{\texttt{rname,rnations},\ldots}$
$\texttt{RNCOP}_{Top}$
$⟕_{\texttt{ncusts=nlabel}}$
$\pi_{\texttt{rlabel},\ldots}$
$\texttt{rnations}_{Dict}$
$⟕_{\texttt{corders=clabel}}$
$\pi_{\texttt{nlabel,corders}}$
$\texttt{ncusts}_{Dict}$
$⟕_{\texttt{oparts=olabel}}$
$\pi_{\texttt{clabel,oparts}}$
$\texttt{corders}_{Dict}$
$\Gamma^{+\,\texttt{qty*price}}_{\texttt{olabel,pname},\ldots}$
$\bowtie_{\texttt{pid}}$
$\pi_{\texttt{olabel,pid,qty},\ldots}$
$\pi_{\texttt{pname,price}}$
$\texttt{oparts}_{Dict}$
Part

2 Biomedical Query Benchmark

This section contains details of the biomedical benchmark that was developed in collaboration with a precision medicine start-up. The biomedical benchmark is a collection of NRC queries that perform multiomic analyses, including an end-to-end pipeline E2E that is based on an analysis that uses several genomic datasets to identify driver genes in cancer[21]. Given that cancer progression is determined by the accumulation of mutations and other genomic aberrations within a sample [4], this analysis integrates somatic mutations, copy number information, protein-protein interactions and gene expression data. The benchmark also includes three queries reflecting web-based exploratory analysis that occurs through clinical user interfaces [12]. The queries are nested-to-nested, representing a scenario where the clinician wants to explore nested data results, and each query applying an additional operation on the next. C_1 groups BN_2 to return a three-level nested output. C_2 joins BF_2 at level 1 of BN_2 then groups as in C_1. C_3 proceeds the same way and aggregates the result of the join prior to grouping. This section continues with details on these datasets and then describes the queries of the analysis and additional queries.

2.1 Inputs

This section explains the inputs used within the biomedical benchmark. The majority of the datasets are provided from the Genomic Data Commons (GDC) [10], which houses public datasets associated with the International Cancer Genome Consortium (ICGC). The types described below are often truncated for simplicity. The inputs include a two-level nested relation BN_2 (280GB) [13, 14], a one-level nested relation BN_1 (4GB) [19], and five relational inputs - the most notable of which are BF_1 (23G), BF_2 (34GB), and BF_3 (5KB) [13, 8]. Full details of these queries can be found in the code repository [7].

2.1.1 BN_2: Occurrences

An occurrence is a single, somatic mutation belonging to a single sample that has been annotated with candidate gene information. *Somatic mutations* are cancer-specific mutations that are identified within each sample, and are identified by comparing a sample's cancerous genome to a non-cancerous, reference genome. Note that the term mutation is often used interchangeably with variant. *Candidate genes* are assigned to mutations based on proximity of a given mutation to a gene on the reference genome. In a naive assignment, a reference gene is a candidate if the mutation lies directly upstream, downstream, or on a gene; however, mutations have been shown to form long-range functional connections with genes [18] and as such candidacy can best be assigned based on a larger flanking region of the genome. With this in mind, the same gene could be considered a candidate gene for multiple mutations within a sample.

Variant annotation is the process that assigns candidate genes to every mutation within each sample. A popular annotation tool is the Variant Effect

Predictor (VEP) [14]. The `Occurrences` input is created by associating each simple somatic mutation file (MAF) with nested annotation information from VEP; this returns annotations in JSON format with mutation information at the top-level, a collection of candidate genes for that mutation and corresponding consequence information on the first level, and a collection of additional consequence information on the second level. Each of the mutations within each sample will contain the corresponding nested annotation information.

The tuples in the `candidates` collection contain attributes that correspond to the impact a mutation has on a gene. `impact` is a value from 0 to 1 denoting any known detrimental consequence a mutation has to a candidate gene. `sift` and `poly` are additional impact scores that rely on prediction software [20, 1]. Given that genes code for proteins and proteins have functional consequences that are attributed disease, these scores reflect the predicted role a mutation has in functional changes to proteins based on alterations to a gene sequence. The `consequences` for each candidate gene contain qualitative descriptions of mutation impact to a gene sequence based on a standardized set of categorical descriptions from the sequence ontology (SO) [8].

VEP also takes a distance flag to specify the upstream and downstream range from which to identify gene-based annotations. This flag is used to increase the flanking region of candidate genes associated to each somatic mutation for each sample. From a technical stand point, increasing the distance will increase the size of `candidates` and simultaneously increase the amount of skew.

The type of `Occurrences` is:

$$\begin{aligned}&\mathit{Bag}\,(\langle\, \texttt{sample} : \mathit{string}, \texttt{contig} : \mathit{string}, \texttt{start} : \mathit{int}, \texttt{end} : \mathit{int},\\ &\quad \texttt{reference} : \mathit{string}, \texttt{alternate} : \mathit{string}, \texttt{mutationId} : \mathit{string},\\ &\quad \texttt{candidates} : \mathit{Bag}\,(\langle\, \texttt{gene} : \mathit{string}, \texttt{impact} : \mathit{string},\\ &\qquad \texttt{sift} : \mathit{real}, \texttt{poly} : \mathit{real}, \texttt{consequences} : \mathit{Bag}\,(\langle\, \texttt{conseq} : \mathit{string} \rangle)\,\rangle)\,\rangle).\end{aligned}$$

The shredded representation of `Occurrences` consists of:

- a 6G top-level bag $\texttt{Occurrences}^{\textsc{Flat}}$ of type $\mathit{Bag}(\langle\ \texttt{sample} : \mathit{string}, \texttt{contig} : \mathit{string}, \texttt{start} : \mathit{int}, \texttt{end} : \mathit{int}, \texttt{reference} : \mathit{string}, \texttt{alternate} : \mathit{string}, \texttt{mutationId} : \mathit{string}, \texttt{candidates} : \mathit{Label}\ \rangle)$,
- for `candidates` labels, a a 281G dictionary $\texttt{Occurrences}^{\textsc{Dict}}_{\texttt{candidates}}$ of type $\mathit{Label} \to \mathit{Bag}(\langle\ \texttt{gene} : \mathit{string}, \texttt{impact} : \mathit{real}, \texttt{sift} : \mathit{real}, \texttt{poly} : \mathit{real}, \texttt{consequences} : \mathit{Label}\ \rangle)$, and
- for `consequences` labels, a 35G dictionary $\texttt{Occurrences}^{\textsc{Dict}}_{\texttt{candidates\_consequences}}$ of type $\mathit{Label} \to \mathit{Bag}(\langle\ \texttt{conseq} : \mathit{string} \rangle)$.

□

2.1.2 BF$_2$: Copy Number

Copy number values correspond to the amplification or deamplification of a gene for each sample, and are also found by comparing against non-cancerous, reference copy number values. The copy number information is provided per gene. The copy number information is reported for each physical sample taken from a patient; this is denoted `aliquot`. The type of the copy number information is:

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

2.1.3 BN$_1$: Protein-protein Interactions

Protein-protein interaction networks describe the relationship between proteins in a network. This `Network` input is derived from the STRING [19] database. The network is represented with a top-level node tuple and a nested bag of edges, where each edge tuple contains an edge protein and a set of node-edge relationship measurements. The type is:

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

2.1.4 BF$_1$: Gene Expression

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

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

2.1.5 Mapping Files

Sample Metadata. The `Samples` input maps samples to their aliquots; for the sake of this analysis `sample` maps to a patient and `aliquot` associates each biological sample taken from the patient. The type is:

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

BF_3: Sequence Ontology. The `SOImpact` input is a table derived from the sequence ontology [8] that maps a qualitative consequence to a quantitative consequence score (`conseq`). This is a continuous measurement from 0 to 1, with larger values representing more detrimental consequences. The type is:

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

Biomart Gene Map. The `Biomart` input is exported from [17]. It is a map from gene identifiers to protein identifiers. This map is required to associate genes from `Occurrences` and `CopyNumber` to proteins that make up `Network`. The type is:

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

These inputs described in this section are used in the queries described in the next section.

2.2 E2E: Pipeline Queries

The queries of the cancer driver gene analysis are an adaptation of the methods from [21]. They work in pipeline fashion to integrate annotated somatic mutation information (`Occurrences`), copy number variation (`CopyNumber`), protein-protein network (`Network`), and gene expression (`GeneExpression`) data. The idea is to provide an integrated look at the impact cancer has on the underlying biological system. The analysis takes into account the effects a mutation has on a gene, the accumulation of genes with respect to both copy number and expression, and the interaction of genes within the system.

Mutations that play a driving role in cancer often occur at low frequency [11], making cohort analysis across many samples important in their identification. Further, cancer is not just the consequence of a single mutation on a single gene. The interaction between genes in a network, the number of such genes, and their expression levels can provide a more thorough look at cancer progression [3]. The queries below define the analysis. The queries work in pipeline fashion where the materialized output from one query is used as input to a query later on in the pipeline.

The pipeline starts with the integration of mutation and copy number variation to produce a set of hybrid-scores for each sample. The hybrid-scores are then combined with network interactions to determine effect-scores. The effect-scores are further combined with gene expression information to determine the connection scores for each sample. The queries conclude by combining the connection scores across all samples, returning connectivity scores for each gene. The genes with the highest connectivity scores are considered drivers.

2.2.1 Step$_1$: Hybrid scores

The hybrid score query is the first step in the pipeline. A *hybrid-score* is calculated for each candidate gene within a sample by combining mutation impact and copy number information for that sample, thus providing a score that corresponds to the likelihood that the gene is a driver. The output of this step remains grouped by sample in order to continue integrating sample-specific genomic datasets that can further contribute to our understanding of driver genes in cancer in the steps below.

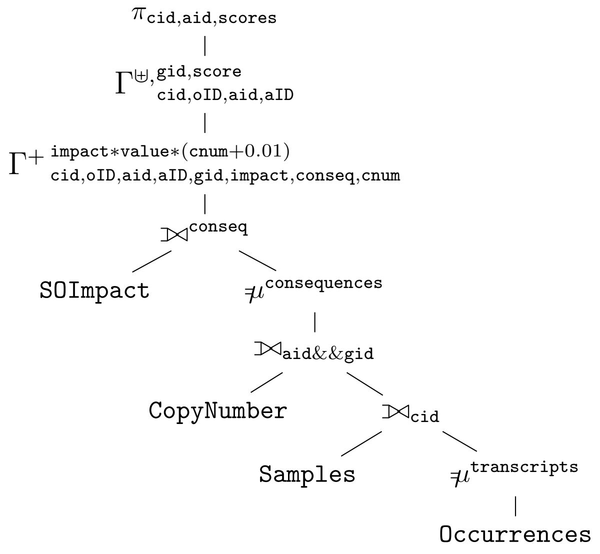
The query below describes the process of creating hybrid scores based on the `Occurrences` input. The main difference between this version and the running example is that `Samples` provides a map between `sample` and `aliquot` used to join `CopyNumber`, and the hybrid score is determined for every `aliquot`. In addition, conditionals are used to assign qualitative scores based on the human-interpretable level of impact (`impact`).

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

The output type of this query is:

$$Bag\,(\langle\,\texttt{sample} : string, \texttt{aliquot} : string, \texttt{scores} : Bag\,(\langle\,\texttt{gene} : string, \texttt{score} : real\rangle)\,\rangle).$$

The plan produced from the standard pipeline is:



$\pi_{\text{cid,aid,scores}}$
$\Gamma^{\uplus,\text{gid,score}}_{\text{cid,oID,aid,aID}}$
$\Gamma^{+\ \text{impact}*\text{value}*(\text{cnum}+0.01)}_{\text{cid,oID,aid,aID,gid,impact,conseq,cnum}}$
$⟕^{\text{conseq}}$
SOImpact
$\mu^{\text{consequences}}$
$⟕_{\text{aid\&\&gid}}$
CopyNumber
$⟕_{\text{cid}}$
Samples
$\mu^{\text{transcripts}}$
Occurrences

2.2.2 Step$_2$: By Sample Network

The second step in the pipeline aggregates on an individual's sample network, based on the hybrid scores. The goal is to associate each gene in the nested `edges` collection of `Network` with the corresponding hybrid scores for a sample. For each gene in this collection, the product of the hybrid score and the relationship measurement for that edge (`distance`) are summed for each node in the network for each sample.

```
SampleNetwork ⇐
for h in HybridMatrix union
  {⟨sample := h.sample, aliquot := h.aliquot, nodes :=
    sumBy$^{\text{score}}_{\text{gene}}$(
      for n in Network union
        for e in n.edges union
          for b in Biomart union
            if e.edgeProtein == b.protein then
              {⟨nodeProtein := n.nodeProtein, score := e.distance ∗ h.hscore⟩})⟩}
```

The output type of this query is:

$$Bag\,(\langle\,\texttt{sample} : string, \texttt{aliquot} : string, \texttt{nodes} : Bag\,(\langle\,\texttt{nodeProtein} : string, \texttt{score} : real\rangle)\,\rangle).$$

2.2.3 Step$_3$: Effect scores

The effect scores are calculated using the materialized output of the previous two steps, denoted `SampleNetwork` and `HybridMatrix`. The nested `nodes` collection of `SampleNetwork` contains the sum of the combined gene interaction and hybrid score across the `edges` collection for each node gene in the top-level tuples of `Network`. These values are then combined with the hybrid scores for each node gene to produce the effect matrix.

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

The output type is:

$$Bag\,(\langle\,\texttt{sample} : \textit{string}, \texttt{aliquot} : \textit{string}, \texttt{scores} : Bag\,(\langle\,\texttt{gene} : \textit{string}, \texttt{score} : \textit{real}\rangle)\,\rangle).$$

Step$_4$: Connection scores. Connection scores are determined by combining the effect scores and gene expression data. Gene expression data uses the normalized count (FPKM) measurement discussed in the input section above.

```
ConnectMatrix ⇐
for s in EffectMatrix union
  {⟨ sample := e.sample, aliquot := e.aliquot, scores :=
    $\texttt{sumBy}^{\texttt{score}}_{\texttt{gene}}$(
      for e in s.scores union
        for g in GeneExpression union
          if e.gene == g.gene then
            {⟨ gene := e.gene, score := e.score ∗ g.fpkm ⟩}) ⟩}
```

The output type is:

$$Bag\,(\langle\,\texttt{sample} : \textit{string}, \texttt{aliquot} : \textit{string}, \texttt{scores} : Bag\,(\langle\,\texttt{gene} : \textit{string}, \texttt{score} : \textit{real}\rangle)\,\rangle).$$

2.2.4 Step$_5$: Gene connectivity

The gene connectivity sums up connection scores for each gene across all samples. The genes with the highest connection scores are taken to be drivers.

```
Connectivity ⇐
sumBy$^{score}_{gene}$(
for $s$ in ConnectMatrix union
  for $c$ in $s$.scores union
    {⟨gene := $c$.gene, score := $s$.score⟩})
```

The output type is:

$$Bag\ (\langle \texttt{gene} : string, \texttt{score} : real \rangle).$$

2.3 Clinical exploration queries

The queries reflect requests a clinician may make from a user-interface; for example, an electronic health record system that provides access to `Occurrences` and `CopyNumber`. The queries contain a combination of restructuring, nested joins, and aggregation.

2.3.1 C$_1$: Group occurrences by sample

This query groups occurrences by sample producing a bag of nested mutation information for each sample. The query also associates a quantitative value to the consequences at the lowest level in the process. The output has four levels of nesting.

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

2.3.2 C$_2$: Integrate copy number and occurrences, group by sample

This query is similar to above, but joins copy number data on the second level - per each gene - while constructing mutation groups per sample.

```
OccurCNVJoin ⇐
for $s$ in Samples union
  {⟨ sample := $s$.sample, mutations :=
    for $o$ in Occurrences union
      if $s$.sample == $o$.sample then
      {⟨ mutationId := $o$.mutationId, ..., candidates :=
        for $t$ in $o$.candidates union
          for $g$ in SOImpact union
            if $g$.gene == $t$.gene then
            {⟨ gene := $t$.gene, cnum := $g$.cnum, ..., consequences :=
              for $c$ in $t$.consequences union
                for $i$ in SOImpact union
                if $c$.conseq == $i$.conseq then
                {⟨ conseq := $i$.conseq, score := $i$.value ⟩} ⟩} ⟩} ⟩}
```

2.3.3 C$_3$: Aggregate copy number and occurrences, group by sample

This query groups by sample, joins copy number at the second level, joins quantitative consequence values at the third level, and aggregates the product of copy number and consequence score for each gene.

```
OccurCNVAgg ⇐
for $s$ in Samples union
  {⟨ sample := $s$.sample, mutations :=
    for $o$ in Occurrences union
      if $s$.sample == $o$.sample then
      {⟨ mutationId := $o$.mutationId, ..., candidates :=
        $\text{sumBy}^{\text{score}}_{\text{gene}}$(
        for $t$ in $o$.candidates union
          for $g$ in SOImpact union
            if $g$.gene == $t$.gene then
              for $c$ in $t$.consequences union
                for $i$ in SOImpact union
                if $c$.conseq == $i$.conseq then
                {⟨ gene := $t$.gene, score := $c$.cnum $*$ $i$.value ⟩}) ⟩} ⟩}
```

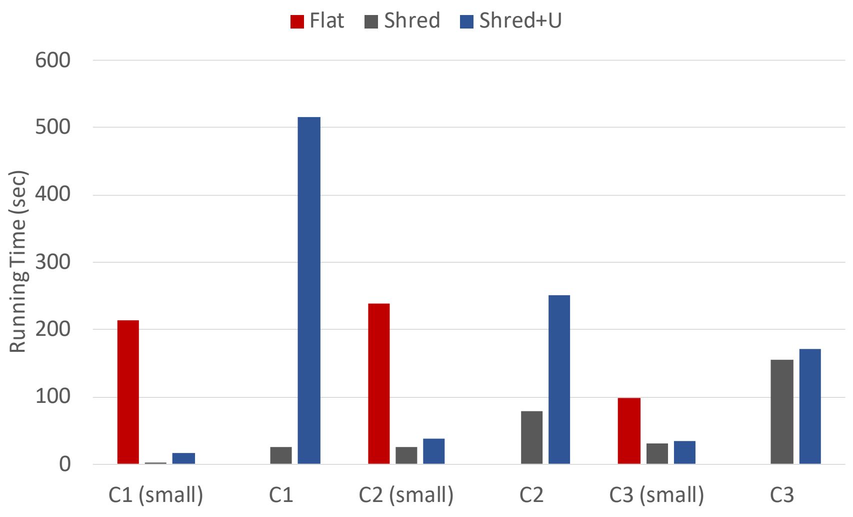


Figure 1: Results for the clinical exploration queries.

2.3.4 Performance

The clinical exploration queries were evaluated using varying sizes of the `Occurrences` input; one small collection based on a 168M mutation file (identified by small) and a large collection of 42G annotated mutations. *Note this is a smaller dataset that is used in the* E2E. Figure 1 displays these results. STD was unable to run to completion for the larger input for all queries, overloading the available memory on the system each time. The shredded pipeline was able to complete for all queries exhibiting resilience by distributing the large inner collections.

The clinical queries extend the benchmark to explore how the performance of generic queries that do not necessarily fit into a whole analysis pipeline. The lack of projections and integration with other datasets mimic the behavior of a query that returns all data to display in a user-interface, for example.

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]`

$=\!\!\!\mu^{a}\ (X)$	`// 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)`
$X ⟕_{f(x)=g(y)} Y$	`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 2: Plan language operators and their semantics for Spark Datasets.

3 Spark Implementation of the Plan Operators

Figure 2 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 $=\!\!\!\mu^{a}\ (X)$, 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 3 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 RDD API
$\sigma_{p(x)}(X)$	`X.filter(x => p(x))`
$\pi_{a_1,\dots,a_k}(X)$	X.map(x => R(x.a_1, ..., x.a_k))

$\mu^{a_i}(X)$	`// x.drop(a) excludes attribute a from tuple x`
	`X.flatMap(x => x.a.map(y => (x.drop(a), y)))`

$=\!\!\mu^{a}\ (X)$	`// 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`

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

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

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

Figure 3: Plan language operators and their semantics for Spark RDDs.

4 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:

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

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:

```
$\texttt{LabDomain}_{\sigma_0\ a_i,\tau'} \Leftarrow$
  for $t$ in $\texttt{MatDict}_{\sigma,\tau}$ union
    if $(\texttt{ExtrctTag}(t) == \tau)$ 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.

5 Additional Experimental Results

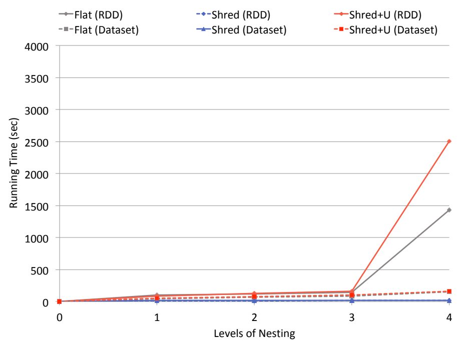
This section contains additional experimental results. All results are for 100GB of non-skewed TPC-H data (scale factor 100, skew factor 0). We use the flat-to-nested and nested-to-nested queries to compare the performance of the standard and shredded pipeline using both RDDs and Datasets. As in the body of the paper, we use Std (standard pipeline), Shred (shredded pipeline without unshredding), and Unshred (shredded pipeline with unshredding) to represent runs from our framework. 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.

5.1 Spark RDDs vs Spark Datasets

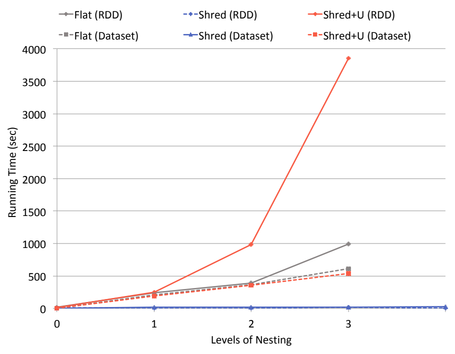
The flat-to-nested and nested-to-nested queries are used to compare the performance of Std (standard pipeline), Shred (shredded pipeline without unshredding), and Unshred (shredded pipeline with unshredding) using RDDs and Datasets. Figure 4 displays the results for the flat-to-nested queries with (4a) and without projections (4b). 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 Unshred and Std 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, Unshred with RDDs shows increasingly worse performance starting at two levels of nesting. Shred with RDDs and Shred with Datasets have similar performance.

As stated in Section 1.1, the plan produced for unshredding in the shredded pipeline and the plan for the standard pipeline are identical; thus, the difference tin performance is attributed to code generation for the unshredding procedure. Both methods use a series of cogroups to build up a nested set; however, unshredding requires additional map operations and intermediate object creation that are required for reconstructing nested objects from dictionaries. Case classes that lack binary encoders require a significant amount of time and space to create and store, which is a cost that only increases with the levels of nesting. Unshred 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 Std , 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 2 to the project operator in Figure 3. 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



(a) Narrow schema



(b) Wide schema

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

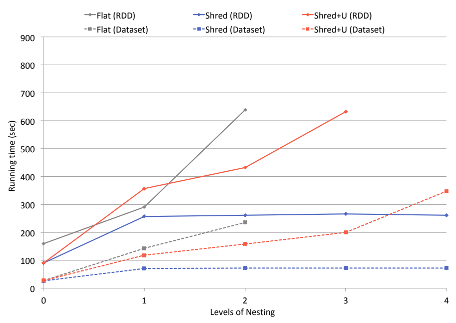
explicit map operation allowing for further performance benefits from the Spark optimizer. Further, when the time comes to construct `R` objects, the Dataset API will leverage the binary format and incur a much smaller memory footprint.

Figure 5 further highlights the benefits of Datasets with nested-to-nested queries. Both with and without projections, the difference between SHRED with RDDs and SHRED with Datasets shows a 2× performance improvement of the explicit statement of attributes within the Dataset API, and STD has decreased performance with RDDs. While the unnest operators used in the code generators both use flat-map operations, the Dataset API maintains a low-memory footprint for the newly created objects (`.as[R]` in the code generator).

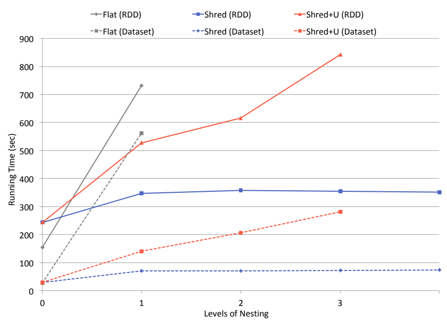
These results show that code generation with Datasets has minimized overhead in object creation and gains further improvements from passing meta-information to the Spark optimizer. Beyond the application to Spark, these results should be useful for further implementations of automated nested query processing on distributed systems.

5.2 Additional Competitors

We used the TPC-H benchmark to compare to a wide array of external competitors: an implementation via encoding in SparkSQL [2]; Citus, a distributed version of Postgres [5]; MongoDB [16], and the recently-developed nested relational engine DIQL [9]. SparkSQL outperformed the other competitors, so those results were included in the body of the paper; this section shows the extended results, including all competitors, for the flat-to-nested, nested-to-nested, and nested-to-flat queries. The source code for each of the queries are available in the github.



(a) Narrow schema



(b) Wide schema

Figure 5: Performance comparison of RDDs verse Datasets for the nested-to-nested benchmarked queries.

Evaluation strategies and competitors. We explored several potential competitors for use in the comparison. The following competitors were able to perform at least one of the TPC-H benchmark queries, and thus are represented in the subsequent results.

- *SparkSQL*:
  The SparkSQL queries were manually written based on two restrictions. First, SparkSQL does not support explode (i.e., UNNEST) operations in the SELECT clause, requiring the operator to be kept with the source relation which forces flattening for queries that take nested input. Second, an (outer) join cannot follow an explode statement; this means the query must be written in the following form:

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

- *DIQL*:
  The syntax of DIQL fully supports all the queries in the TPC-H benchmark; however, this is an experimental system and we uncovered bugs during this process. With this in mind, we provide the results for the flat-to-nested queries only. The DIQL Spark API has slightly different system requirements and we were only able to compile and run the queries with Spark 2.4.3 and Scala 2.11.

- *Postgres+Citus*:

We use a distributed version of Postgres (Citus) as the representative relational database engine. We use a coordinator Postgres instance with five workers, exactly like the Spark. We cached inputs and parallelized processes as much as possible with the following:

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

However, the results using default values had better performance. The Citus queries were manually written, using both arrays and JSON with and without caching inputs. We report the runtimes of the array based queries, without caching inputs, and default worker configurations since this continuously outperformed the others.

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

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

The result of the subquery (t1 join t2) will be collected entirely at the master and then partitioned to workers according to the next join key. This is obviously inefficient and has restrictions logged in Citus as: *DETAIL: Citus restricts the size of intermediate results of complex subqueries and CTEs to avoid accidentally pulling large result sets into once place.* Third, left outer joins between tables partitioned on different keys are not yet supported (https://github.com/citusdata/citus/issues/2321). Finally, to avoid pulling data back to master and enable outer joins between relations partitioned on different keys, we manually created execution plans where at eachstep we (outer) join two relations partitioned on the same key and write the result back into a distributed materialized view partitioned on the next join key in sequence. That means we had to materialize the entire flattened nested object to get it repartitioned by partkey before joining with Part. Each nested-to-flat requires 2 queries, while each nested-to-nested has one extra query for the final regrouping.

- *MongoDB*:
  We use MongoDB with one master and five workers, as in the Spark and Citus setup. The queries were hand-written based on the following restrictions. Only one collection can be sharded when performing lookups (joins), the inner one must be local. The only join strategy is to iterate (in parallel) over the outer collection and do lookups on the inner collection, which is located on one machine; thus, this is a bottleneck. We find that MongoDB has good with selective filters over a single collection, not designed for queries over multiple collections or even single-collection queries that return many documents. Nested collections formed using the `$push` accumulator are currently capped at 100MB; pipelines using more than 100MB will fail.

Additional competitors explored. The following systems were also explored, but were unable to support the queries of the benchmark.

- *Rumble*:
  Rumble transforms JSONiq to Spark and supports local and distributed execution. We discovered problems running even toy examples doing data denormalization. Initially, outer joins were not supported: `https://github.com/RumbleDB/rumble/issues/760`. We reported this and it was fixed, but now outer joins with distributed collections are transformed into Cartesian products. In general, the Rumble JSONiq language is not providing several of the operations available in Spark (e.g., caching, schema handling) are not available through their JSONiq language.

- *Zorba*:
  Zorba (JSONiq) has no support for distributed execution, and has not been maintained in the past 4 years [22].

- *MonetDB*:
  MonetDB provides no support for array type and array operations. The system does supports JSON operations over strings, but there is no easy way to transform tables to JSON objects, and manually creating JSON strings throws errors [15].

- *Cockroach*:
  CockroachDB does not support nested arrays or ordering by arrays [6].

- *VoltDB*:
  VoltDB does not support for arrays. There is support for JSON, but is is up to the application to perform the conversion from an in-memory structure to the textual representation. In addition, there is a size limit for JSON values. The VARCHAR columns used to store JSON values are limited to one megabyte (1048576 bytes). JSON support allows for augmentation of the existing relational model with VoltDB; however, it is not

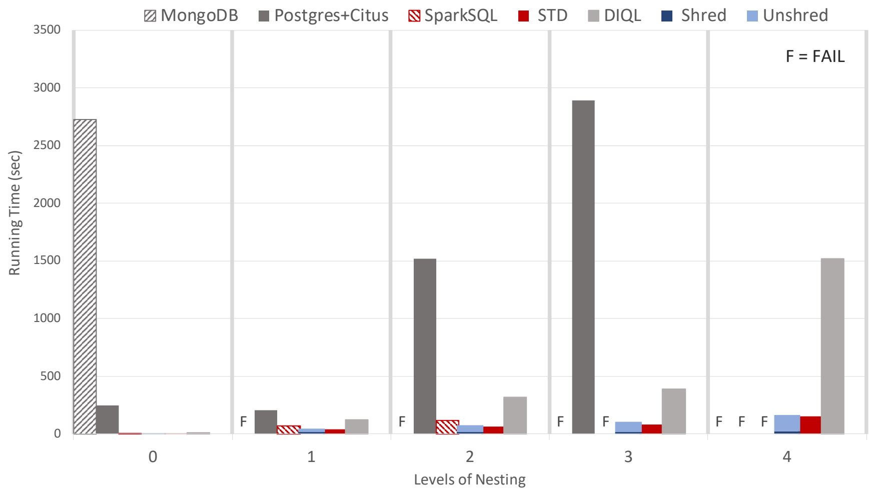
intended or appropriate as a replacement for pure blob-oriented document stores.

- *BerkeleyDB*:
  BerkeleyDB is a key-value store with a SQL layer on top, similar in a way to SQLite.

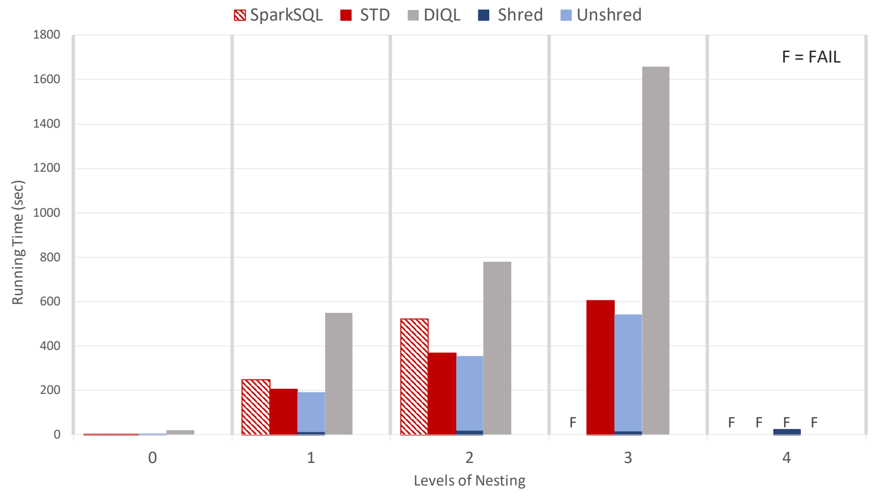
- *YugabyteDB*:
  YugabyteDB seemed like a good candidate as it supports distributed execution and much of SQL, but the performance was too poor to explore further. For example, the following query took four minutes with 18760 orders tuples and 2500 user tuples:

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

Flat-to-nested for non-skewed data. Figure 6a displays the results for MongoDB, Postgres Citus, DIQL, SparkSQL, Std , Shred, and Unshred for the narrow flat-to-nested queries of the TPC-H benchmark. Given the performance for all systems is worse for wide tuples, we did not explore the performance of MongoDB and Citus for the wide variants. Figure 6b displays the results for DIQL, SparkSQL, Std , Shred, and Unshred for the wide flat-to-nested queries.



(a) Narrow schema



(b) Wide schema

Figure 6: Performance comparison of flat-to-nested queries including all competitors.

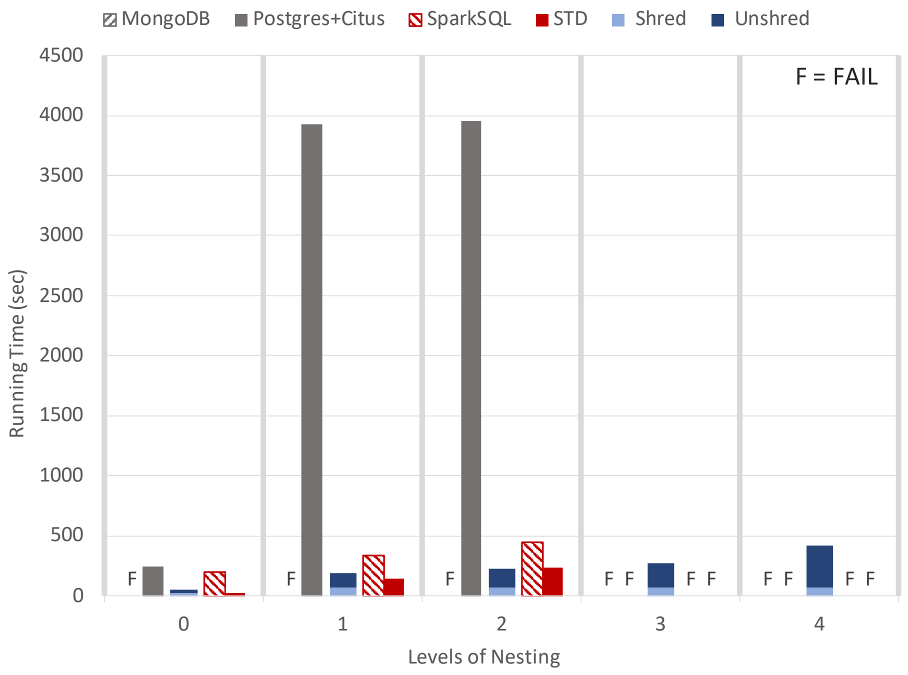


Figure 7: Performance comparison of narrow nested-to-nested queries including all competitors.

Nested-to-nested for non-skewed data. Figure 7 displays the results for MongoDB, Postgres Citus, SparkSQL, Std , Shred, and Unshred for the narrow nested-to-nested queries of the TPC-H benchmark. Given the performance for all systems is worse for wide tuples, we did not explore the performance of MongoDB and Citus for the wide variants; thus, the wide variants for SparkSQL, Std , Shred, and Unshred can be found in the main body of the paper.

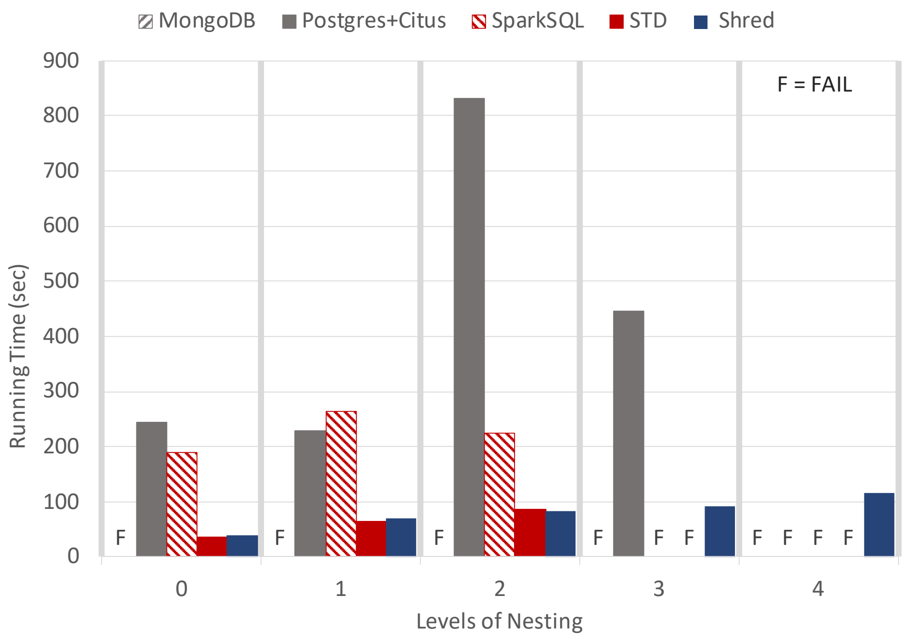


Figure 8: Performance comparison of narrow nested-to-flat queries including all competitors.

Nested-to-flat for non-skewed data. Figure 8 displays the results for MongoDB, Postgres Citus, SparkSQL, STD , SHRED, and UNSHRED for the narrow nested-to-flat queries of the TPC-H benchmark. As with the previous two query categories, we provide only the narrow variants for MongoDB and Postgres Citus. The wide variants for SparkSQL, STD , SHRED, and UNSHRED can be found in the main body of the paper.

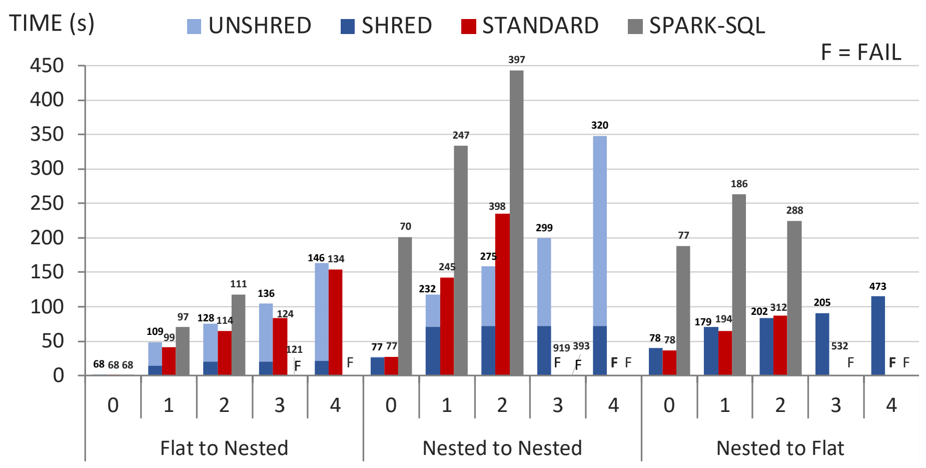


Figure 9: Performance comparison of narrow TPC-H benchmark queries with total shuffle memory (GB).

5.3 Total shuffle for non-skewed TPC-H benchmark

Figure 9 and Figure 10 provides annotated results for Figure 7 in Section 7 of paper, which includes the total shuffled memory (GB) for each run. If a job crashes at a particular nesting level, we do not report any further total shuffle memory.

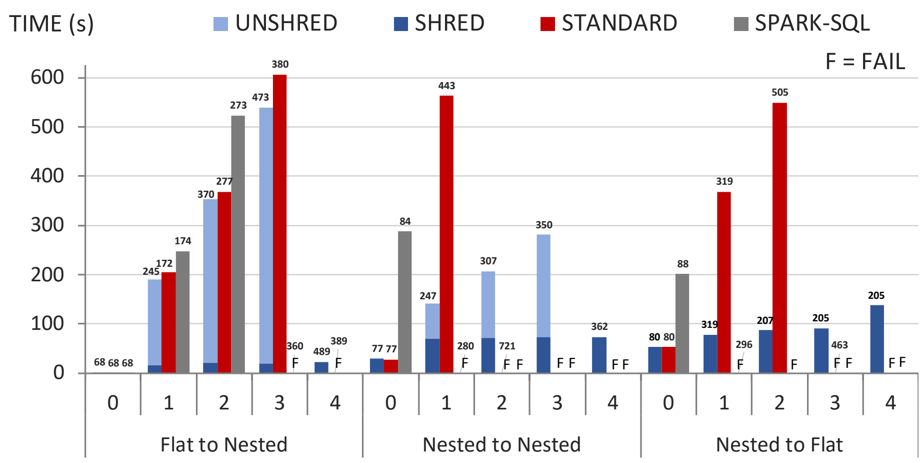
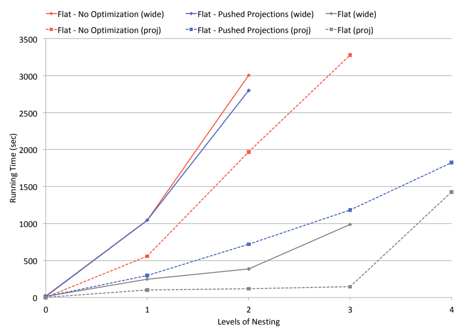


Figure 10: Performance comparison of wide TPC-H benchmark queries with total shuffle memory (GB).

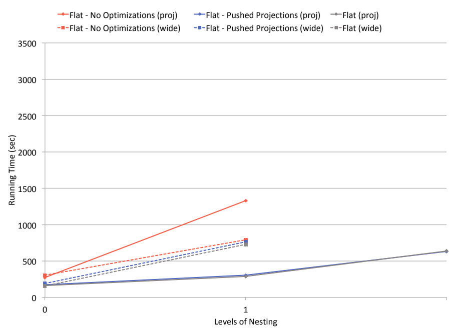
5.4 Standard pipeline optimizations

This experiment highlights how the framework can leverage database-style optimizations to automatically generate programs that are comparable to hand-optimized programs. Plans are generated using the standard pipeline with increasing amounts of optimizations applied to both the flat-to-nested and nested-to-nested queries. Figure 11 shows the results of this experiment. STD with no optimizations is the untouched plan that comes out of the unnesting algorithm. STD with pushed projections is the plan from the unnesting algorithm with projections pushed. STD is the standard pipeline used in all experiments with projections pushed, nests pushed past joins, and merged into cogroups where relevant. STD 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 11a 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 the experiments in the previous sections have shown that the performance of STD is heavily impacted by the presence of projections (ie. the number of attributes an output tuple). For nested-to-nested queries, STD is the only strategy to survive past one level of nesting. These results show that database-style optimizations are not only beneficial to improve performance, but are necessary when using flattening methods even with shallow-nested objects.



(a) Flat to Nested



(b) Nested to Nested

Figure 11: Performance comparison of benchmarked queries for increasing optimization levels of the standard pipeline.

5.5 Additional Skew-handling results

`COP` shuffle in skew-handling results. Figure 12 shows the amount of data shuffled from `COP` prior to the nested join with `Part` for the nested-to-nested TPC-H query used in the skew-handling results in the paper. The results highlight how the skew-aware shredded pipeline leads to less than a gigabyte of shuffling for moderate and high levels of skew. There are no `COP` shuffle results for the standard pipeline due to the application failing during flattening. SparkSQL survives flattening for high levels of skew, but fails while performing the join with `Part`. The shuffling of the standard pipeline shows that at lower levels of skew the local aggregation is beneficial. At higher levels of skew the local aggregation reduces the data in the skew-unaware pipeline to about 4.5G; however, the skew-aware pipeline has reduced this to only megabytes of data leading to 74x less shuffle than the skew-unaware pipeline.

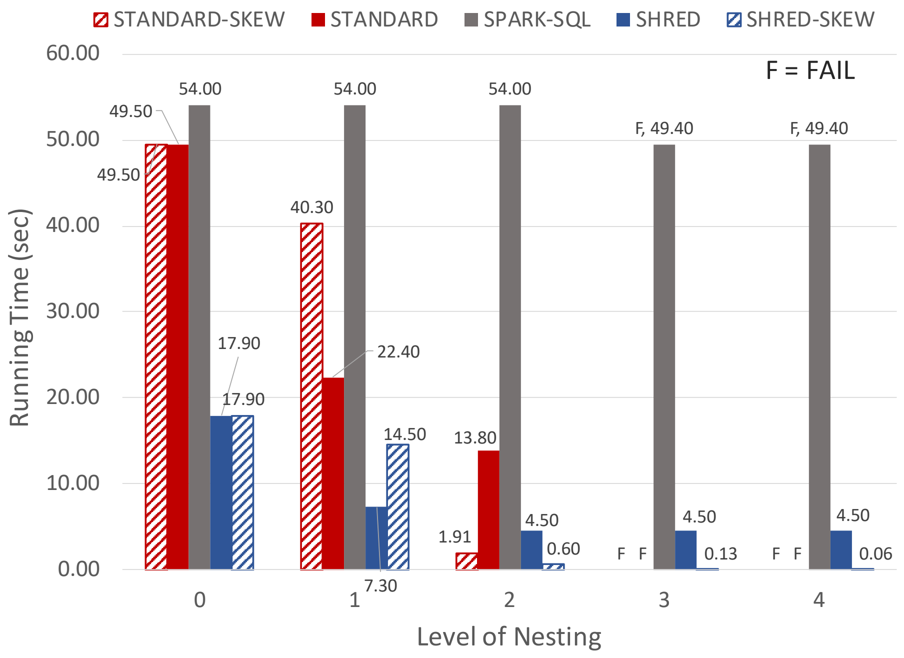


Figure 12: Amount of shuffled data from `COP` prior to joining with `Part` in level 2, narrow, nested-to-nested TPC-H query for skew-aware and skew-unaware pipelines, as well as SparkSQL.

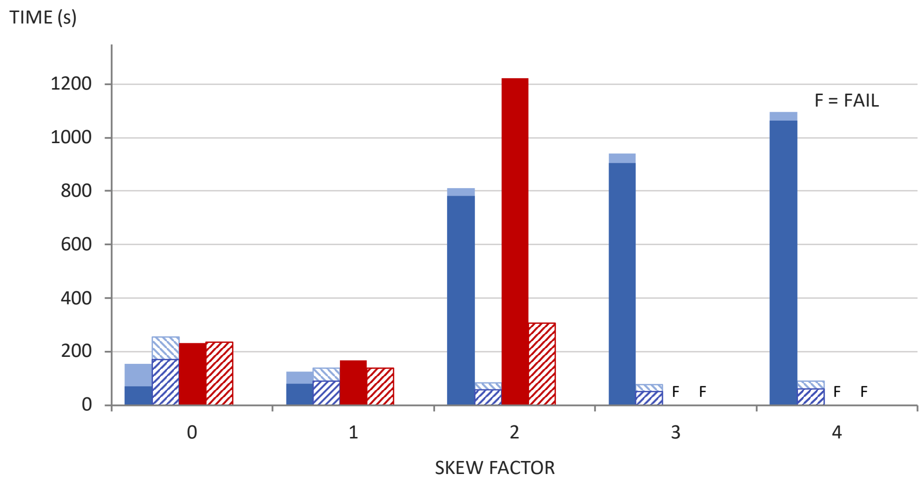


Figure 13: Performance comparison of skew-aware and non-skew aware standard and shredded pipeline without local aggregation.

Without local aggregation. Figure 13 shows the runtimes for the skew-aware and skew-unaware standard and shredded pipeline when local aggregation is disabled.

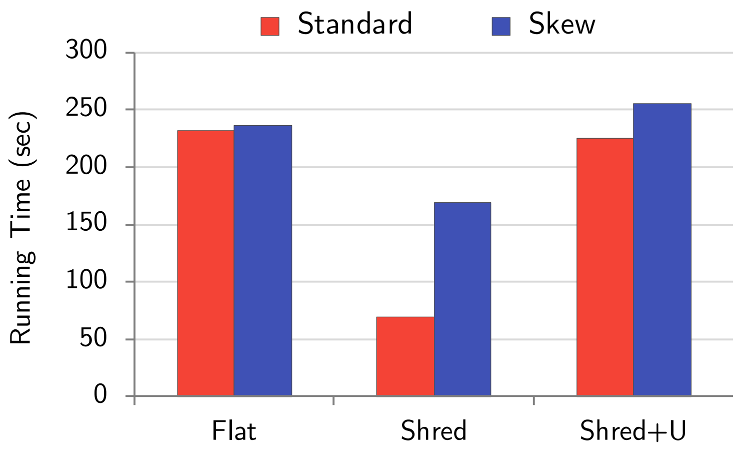


Figure 14: Skew-handling overhead for all strategies.

5.6 Overhead of Skew-Handling

Figure 14 shows the overhead of skew-handling for *non-skewed* data. $\text{Shred}_{\text{skew}}$ exhibits the largest overhead for heavy key collection. Both $\text{Flat}_{\text{skew}}$ and $\text{Shred}_{\text{skew}}$ calculate heavy keys for `Lineitem` $\bowtie$ `Part`, which takes about 12s. The main overhead of $\text{Shred}_{\text{skew}}$ is the heavy key calculation within the final casting of the lowest-level dictionary with `BagToDict`. Unshredding does no heavy key calculations, thus there is no additional overhead for $\text{Shred}^{+U}_{\text{skew}}$.

References

[1] I. A. Adzhubei, S. Schmidt, L. Peshkin, V. E. Ramensky, A. Gerasimova, P. Bork, A. S. Kondrashov, and S. R. Sunyaev. A method and server for predicting damaging missense mutations. *Nature Methods*, 7(4):248–249, 2010.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[3] F. Cheng, J. Zhao, and Z. Zhao. Advances in computational approaches for prioritizing driver mutations and significantly mutated genes in cancer genomes. *Briefings in Bioinformatics*, 17(4):642–656, 08 2015.

[4] Z. Z. Cheng F, Zhao J. Advances in computational approaches for prioritizing driver mutations and significantly mutated genes in cancer genomes. *Briefings in Bioinformatics*, 17(4):642–656, 2016.

[5] Citus, 2020. `www.citusdata.com`.

[6] Cockroach, 2020. `https://github.com/cockroachdb/cockroach`.

[7] Scalable querying of nested data, 2020. `github.com/jacmarjorie/shredder`.

[8] K. Eilbeck, S. E. Lewis, C. J. Mungall, M. Yandell, L. Stein, R. Durbin, and M. Ashburner. The sequence ontology: A tool for the unification of genome annotations. *Nature Methods*, 6:R44, 2005.

[9] L. Fegaras. Compile-time query optimization for big data analytics. *Open Journal of Big Data (OJBD)*, 5(1):35–61, 2019.

[10] Genomic data commons endpoints, 2020. `https://docs.gdc.cancer.gov/API`.

[11] C. Greenman, P. Stephens, R. Smith, G. L. Dalgliesh, C. Hunter, G. Bignell, H. Davies, J. Teague, A. Butler, C. Stevens, S. Edkins, S. O'Meara, I. Vastrik, E. E. Schmidt, T. Avis, S. Barthorpe, G. Bhamra, G. Buck, B. Choudhury, J. Clements, J. Cole, E. Dicks, S. Forbes, K. Gray, K. Halliday, R. Harrison, K. Hills, J. Hinton, A. Jenkinson, D. Jones, A. Menzies, T. Mironenko, J. Perry, K. Raine, D. Richardson, R. Shepherd, A. Small, C. Tofts, J. Varian, T. Webb, S. West, S. Widaa, A. Yates, D. P. Cahill, D. N. Louis, P. Goldstraw, A. G. Nicholson, F. Brasseur, L. Looijenga, B. L. Weber, Y.-E. Chiew, A. deFazio, M. F. Greaves, A. R. Green, P. Campbell, E. Birney, D. F. Easton, G. Chenevix-Trench, M.-H. Tan, S. K. Khoo, B. T. Teh, S. T. Yuen, S. Y. Leung, R. Wooster, P. A. Futreal, and M. R. Stratton. Patterns of somatic mutation in human cancer genomes. *Nature*, 446(7132):153–158, 2007.

[12] i2b2, 2020. `i2b2.org/software/index.html`.

[13] International cancer genome consortium, 2020. `https://icgc.org/`.

[14] W. McLaren, L. Gil, S. E. Hunt, H. S. Riat, G. R. S. Ritchie, A. Thormann, P. Flicek, and F. Cunningham. The ensembl variant effect predictor. *Genome Biology*, 17(1):122, 2016.

[15] MonetDB, 2020. `https://www.monetdb.org/`.

[16] MongoDB, 2020. `www.mongodb.com`.

[17] D. Smedley. The biomart community portal: an innovative alternative to large, centralized data repositories. *Nucleic Acids Research*, 43(W1):W589–W598, 04 2015.

[18] S. Smemo, J. J. Tena, and M. A. Nbrega. Obesity-associated variants within fto form long-range functional connections with irx3. *Nature*, 507(7492):371375, 2014.

[19] D. Szklarczyk, A. Gable, and D. e. a. Lyon. String v11: protein-protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic Acids Res*, 47(D1):D607–D613, 2019.

[20] R. Vaser, S. Adusumalli, S. N. Leng, M. Sikic, and P. Ng. Sift missense predictions for genomes. *Nature Protocols*, 11(1):10731081, 2009.

[21] W. Zhang and S.-L. Wang. A novel method for identifying the potential cancer driver genes based on molecular data integration. *Biochemical Genetics*, 58(1):16–39, 2020.

[22] Zobra, 2016. `https://github.com/zorba-processor/zorba`.