TraNCE: Transforming Nested Collections Efficiently

Citation for published version:
Smith, J, Benedikt, M, Moore, B & Nikolic, M 2021, 'TraNCE: Transforming Nested Collections Efficiently', *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, no. 12, pp. 2727-2730.
https://doi.org/10.14778/3476311.3476330

Digital Object Identifier (DOI):
10.14778/3476311.3476330

Link:
Link to publication record in Edinburgh Research Explorer

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

Published In:
Proceedings of the VLDB Endowment (PVLDB)

TraNCE: Transforming Nested Collections Efficiently

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

Milos Nikolic
University of Edinburgh
milos.nikolic@ed.ac.uk

ABSTRACT

Nested relational query languages have long been seen as an attractive tool for scenarios involving large hierarchical datasets. There has been a resurgence of interest in nested relational languages. One driver has been the affinity of these languages for large-scale processing platforms such as Spark and Flink.

This demonstration gives a tour of TraNCE, a new system for processing nested data on top of distributed processing systems. The core innovation of the system is a compiler that processes nested relational queries in a series of transformations; these include variants of two prior techniques, shredding and unnesting, as well as a *materialization* transformation that customizes the way levels of the nested output are generated. The TraNCE platform builds on these techniques by adding components for users to create and visualize queries, as well as data exploration and notebook execution targets to facilitate the construction of large-scale data science applications. The demonstration will both showcase the system from the viewpoint of usability by data scientists and illustrate the data management techniques employed.

PVLDB Reference Format:
Jaclyn Smith, Michael Benedikt, Brandon Moore, and Milos Nikolic. TraNCE: Transforming Nested Collections Efficiently. PVLDB, 14(12): 2727 - 2730, 2021.
doi:10.14778/3476311.3476330

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

1 INTRODUCTION

The nested relational model and the associated query language Nested Relational Calculus (NRC) have been developed since the late 1980's. Recently there has been renewed attention to NRC due to its synergy with the collection APIs of distributed processing platforms, such as Spark and Flink. TraNCE is a recently-developed platform for evaluating nested data queries [9, 10]. The core of TraNCE is a compiler that processes NRC queries in a series of transformations, including variants of the two major prior techniques, shredding [3] and unnesting [4], as well as a *materialization* transformation that controls the manner in which nested outputs are generated.

Nested data is common across many domains; one important application area is biomedical analysis. Modern biomedical analyses operate on and produce datasets in a variety of complex, domain-specific formats. Advances in genomic sequencing, image processing, and standardization of medical data have generated new opportunities for complex, multimodal data integration tasks that identify targeted treatments for both clinical and research applications. We now review two biomedical use cases that deal with complex, large-scale datasets and highlight many challenges common to all data science applications.

Example 1.1. [Clinical exploration] Consider a scenario where a clinician would like to generate a report that supports exploration of mutations identified in each sample with associated gene-based likelihood scores. The scores provide a predictive measurement of a gene being a driver in a specific disease, such as cancer. This report is produced by performing aggregate analysis on genomic data sources along side clinical attributes, such as diagnosis and treatment history.

A simplified view of the clinical data source, `Samples`, is a flat relation with type:

$$\{\langle\, \mathtt{sid} : \mathit{string}, \mathtt{tumorsite} : \mathit{string}, \mathtt{toutcome} : \mathit{string}\,\rangle\}$$

The first genomic data source, `CopyNumber`, contains copy number information for each sample and each gene, and is also a flat relation with type:

$$\{\langle\, \mathtt{sid} : \mathit{string}, \mathtt{gene} : \mathit{string}, \mathtt{cnum} : \mathit{int}\,\rangle\}$$

The third data source, `Occurrences`, contains a collection of sample-based mutations with nested annotation information and has type:

$$\begin{aligned}&\{\langle\, \mathtt{sid} : \mathit{string}, \mathtt{mutId} : \mathit{string}, \mathtt{candidates} :\\ &\quad \{\langle\, \mathtt{gene} : \mathit{string}, \mathtt{impact} : \mathit{string}, \mathtt{sift} : \mathit{real}, \mathtt{poly} : \mathit{real},\\ &\qquad \mathtt{consequences} :\ \{\langle\, \mathtt{conseq} : \mathit{string}\,\rangle\}\rangle\}\rangle\}\end{aligned}$$

The `candidates` attribute identifies a collection of predicted effects a mutation has on a gene, such as `impact`.

The generation of such a report requires a restructuring that associates the `sid` and `gene` attributes above. Since `gene` is a nested attribute of `Occurrences`, the construction of this analysis is not straightforward and clearly far from the expertise of a clinician. In addition to correctly formulating a query, either as code or in some DBMS language, the analysis could require evaluating datasets at petabyte scale.

The above analysis focuses on data restructuring and aggregation tasks. Research-based pipelines bring more complexities, such

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476330

as interfacing with a variety of external software and statistical libraries in an interactive environment.

Example 1.2. [Interactive analysis with external libraries] The clinician from the prior example has explored the outcome of the report and suspects a possible correlation of the likelihood scores with treatment outcome. A request is sent to the medical research department for further exploration. The researcher further aggregates the results of the analysis from Example 1.1 to calculate likelihood scores for each gene and produce feature vectors for each sample. To iterate more quickly, the feature vectors are materialized into an interactive notebook environment, such as Zeppelin. The researcher passes a subset of the data to train a multi-class neural network using the keras learning framework. The model is tested, features are filtered using chi-square tests, and additional modeling strategies are explored interactively within the notebook. After a series of iterative improvements, a high-confidence model is sent to production to use as a new clinical report. The infrastructure described here goes far beyond nested data management.

TraNCE aims at increasing the level of abstraction for data scientists working with nested data. In this demonstration we will: 1) show the system in action on end-to-end data science workflows while showcasing its benefits; 2) illustrate the transformations within the processing pipeline and the impact of different "levers" provided by the system on runtime performance; 3) show how the system can interface with external data science tools.

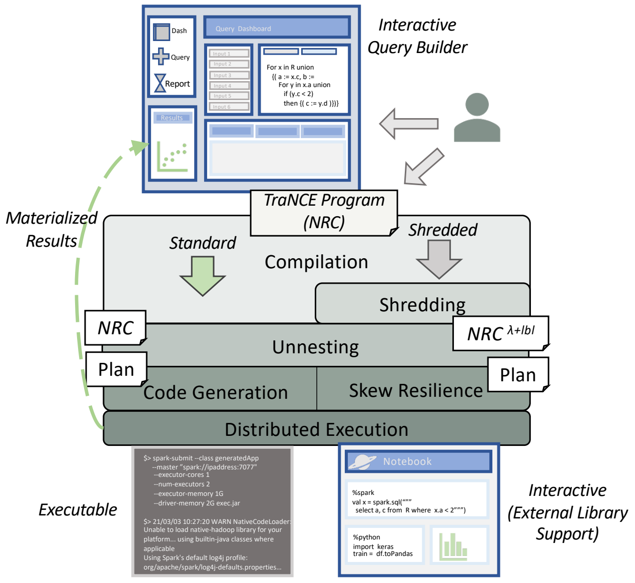
2 RELATED WORK

The advent of big data frameworks have motivated a number of systems that provide higher-level language support compiling to Spark or Flink. Rumble [7] supports the less structured JSON model rather than nested relations, implementing a subset of the proposed JSONiq standard. Diablo [5] focuses on a data model with support for arrays. The compilation pipelines of Rumble and Diablo are quite different from ours, avoiding the use of shredding. Lara and Emma [2, 6] support comprehension-style operations such as those found in NRC, with the aim of assisting the building of machine learning pipelines. But in these approaches the bulk operations are embedded in a host programming language rather than as a standalone declarative dialect. Casper [1] also looks at compiling higher-level code fragments into distributed processing platforms, but aims at general-purpose programming languages. The core algorithms of TraNCE are introduced in [9], and applications to biology are explored in [10]. Shredding and a technique related to unnesting, *lifting*, have been explored in [11, 12], although in those works the target is not a distributed processing platform.

3 TRANCE ARCHITECTURE

The TraNCE architecture is designed to execute NRC queries on top of a distributed processing framework. Figure 1 displays the system architecture, including both frontend and query execution targets. A user can construct NRC queries using a graphical user interface or submit their own NRC queries directly.

When a TraNCE program is submitted to the framework, the source NRC can follow a *standard* or *shredded* compilation route. The standard compilation route uses unnesting techniques, which translates NRC into a query plan that uses bulk operators. The plan



```
For x in R union
  {( a := x.c, b :=
    For y in x.a union
      if (y.c < 2)
      then {( c := y.d )})}
```

```
$> spark-submit --class generatedApp
    --master "spark://ipaddress:7077"
     --executor-cores 1
     --num-executors 2
     --executor-memory 1G
     --driver-memory 2G exec.jar

$> 21/03/03 10:27:20 WARN NativeCodeLoader:
Unable to load native-hadoop library for your
platform... using builtin-java classes where
applicable
Using Spark's default log4j profile:
org/apache/spark/log4j-defaults.properties...
```

```
%spark
val x = spark.sql("""
 select a, c from  R where  x.a < 2""")
```

```
%python
import  keras
train =  df.toPandas
```

Figure 1: TraNCE architecture diagram. User submits queries from a graphical interface or an NRC query. The query is compiled using the standard or shredded compilation route to produce executable Spark/Flink applications, interactive notebooks, or return the results to the graphical interface for further exploration.

is passed to the code generation module to produce an executable Spark or Flink application.

The standard compilation route works on the native, top-level distribution strategy of the processing platform; this strategy can lead to poor data distribution, especially with few top-level tuples or large inner collections. The shredded compilation route optimizes the standard route by operating over a more succinct data representation that admits better distribution of inner collections.

The shredded compilation route adds on additional source-to-source transformations, such as *query shredding* [3]. The structure of the shredded query mimics the structure of the query output, with subqueries for each level. Query shredding in TraNCE is guided by a number of policies, including how to provide the relevant collection identifiers from one level to the next level – a process handled by a transformation called *materialization*. TraNCE supports a "domain-based" materialization policy that gathers identifiers from the prior level via a separate query, and another "input-based" policy that, when applicable, gathers the relevant identifiers from the input dictionaries. The output of materialization is sent to the unnesting module and the compilation proceeds as in the standard route.

Skew is a consequence of the key-based partitioning strategy used in distributed processing platforms; this strategy sends all values with the same key to the same partition. The skew-handling component uses an on-the-fly sampling procedure to identify skew-related bottlenecks and automatically handle distribution of those values at runtime. Given that the shredded representation ensures distribution of inner collections, the shredded compilation method is better suited to handle skew-related issues. While nesting further complicates skew-related issues, skew is a prevalent issue regardless of nesting; thus, both pipelines leverage the skew-handling module to maintain proper distribution of values associated to heavy keys via broadcasting techniques.

Query execution targets are determined by the user. Queries can be compiled and executed within the web interface, returning the materialized result to the front-end data explorer. Alternatively, the user can choose to return the materialized result into a notebook to interact with the data and develop analyses further with external packages. Advanced statistical operations can also be leveraged via user-defined functions that can be applied directly within the TraNCE program; these will be translated into the relevant Scala code in the generated application.

The details of the algorithms are described in [9]. In the demonstration, we aim to show the stages of the compilation pipeline and allow users to explore compilation variants.

4 SCENARIOS

We explain how the high-level scenarios of Section 1 are addressed in our framework.

Clinical exploration. The clinical use case presented in Example 1.1 described returning likelihood scores for each mutation and sample using genomic and clinical information. The user would construct the TraNCE program within the user-interface, first selecting the `Occurrences`, `CopyNumber`, and `Samples` inputs along with the desired projected attributes. The user adds a filter to the data to explore specific tumor origin sites. They build the query with three levels, samples at the top-level with nested mutation information. An aggregate is specified at the lowest level, to produce gene-based likelihood scores for each mutation. At this point, the user has constructed the following source NRC query:

```
ClinicalReport ⇐
  for $s$ in Samples if $s$.tumorsite ∈ {'Breast', 'Lung'} union
  {⟨sid := $s$.sid, toutcome := $s$.toutcome, mutations :=
      for $o$ in Occurrences if $s$.sid == $o$.sid union
      {⟨mutId := $o$.mutId, scores :=
          sumBy$^{\text{score}}_{\text{gene}}$(
             for $t$ in $o$.candidates union
               for $g$ in CopyNumber union
                 if $g$.gene == $t$.gene && $g$.sid == $o$.sid then
                 {⟨gene := $t$.gene, score1 := $t$.sift $*$ $t$.poly,
                   score2 := $t$.impact $*$ ($g$.cnum $+ 0.01$) ⟩}) ⟩} ⟩}
```

The query can be compiled, allowing users to view the intermediate results of compilation. In this example, the standard compilation route would be prohibitively expensive to execute due to burdensome flattening operations; the shredded compilation route exhibits better performance, operating on each level and avoiding expensive flattening procedures. The three levels are translated into three queries; the first two queries are simple operations over `Samples` and the top-level shredded input of `Occurrences`. The third query associates the first-level shredded input of `Occurrences` with `CopyNumber` and applies the `sumBy`; this occurs only on the succinct representation used in shredding, leading to a more lightweight execution than the standard pipeline, which requires flattening, regrouping, and carrying around extra parent attributes during execution. Alternatively, the query can be executed and the results can be navigated within a data explorer. A plot is returned that displays the size of the nested `mutations` attribute for each top-level `sid`. After exploring the data, the user sees that samples with positive response to treatment often have higher likelihood scores than samples with neutral or negative response to treatment. The results of this exploratory analysis can be reported to the research department for further exploration.

Interactive analysis. The researcher doing the analysis aims to build feature sets for each sample to use in a classifier. They use the materialized output of `ClinicalReport` to further aggregate the likelihood scores at the first level and rename the nested attribute to `features`; this produces the following NRC query:

```
Features ⇐
  for $s$ in ClinicalReport union
  {⟨sid := $s$.sid, toutcome := $s$.toutcome, features :=
    sumBy$^{\text{score}}_{\text{gene}}$(
     for $m$ in $s$.mutations union
       for $g$ in $m$.scores union
        {⟨gene := $g$.gene, score := $g$.score1 $*$ $g$.score2⟩})⟩}
```

Upon viewing the compiled plans, the user can see that the `ClinicalReport` query was run using the shredded compilation route, producing a succinct shredded output. They can further transform the output as a nested object without being concerned with the storage format. Results can also be materialized in a notebook, loaded as a Spark DataFrame with the following type:

$$\{\langle\, \mathsf{sid} : \mathit{string}, \mathsf{toutcome} : \mathit{string}, \{\langle \mathsf{gene} : \mathit{string}, \mathsf{score} : \mathit{real}\,\rangle\}\rangle\}$$

At this point, the user interacts with the output of the TraNCE program in the notebook environment. The user first pivots `Features` by `gene` to produce a dataset with columns for `sid`, `toutcome`, and every value of `gene`; the `gene` attributes are the features with the associated `score` as the value. They then employ context switching provided by Zeppelin to represent this Spark DataFrame in Python (`Features.toPandas()`). The data is randomly split into training (70%) and testing (30%) sets using scikit-learn and a neural network is constructed with keras. This is a fully-connected feed-forward multi-class neural network for treatment outcome using all genes. The user can utilize a softmax output, which can be interpreted as a probability distribution over four treatment outcomes: remission, partial, stable, and progressive. The testing phase does not return a high-confidence model, so the user uses scikit-learn to find important features with `SelectKBest` and `chi2`. The top 1600 genes are selected for use in the same model. The results show an overall accuracy of about 45%.

The researcher can then perform a one-vs-rest approach to train binary classifiers for each treatment outcome using the chi-square selected features. The binary models have a sigmoid output that can be interpreted as the probability of a certain treatment outcome corresponding to this model. After training each model, predictions are made using the entire dataset and the computed results are merged. Samples are then classified based on the highest likelihood. The performance of this model is 80%, which is high enough confidence to send the model off to production.

5 DEMONSTRATION WALK-THROUGH

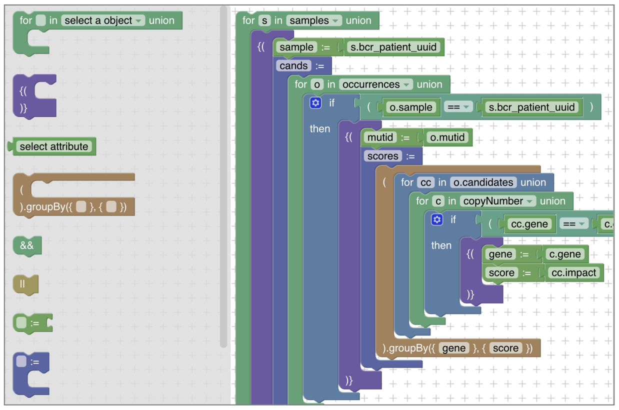
The demonstration showcases TraNCE from several perspectives, including *query builder*, *compilation*, and *results* views.

The query builder view of TraNCE (Figure 2a) enables users to build queries without in-depth knowledge of NRC. The builder uses blockly [8] to provide NRC expression "blocks", which can be pieced together to form a TraNCE program. Inputs are selected within relevant blocks providing the scope for subexpressions. Queries can be imported or deleted using the top right-hand corner menu. An additional dashboard view provides a comprehensive list of all queries; the user can edit, execute, or delete queries from this list.

The compilation view shows the TraNCE compilation process. When compilation is triggered, the compilation window shows the shredded NRC query and the plan produced by the shredded compilation route; the user can also see the standard plan produced without shredding. Mousing over the syntax trees allows the user to see the links between the NRC source and the artifacts produced by compilation. Figure 2b shows the shredded query and plan for Example 1.1. The compilation view also allows the user to see the impact of changes in the materialization and shredding strategy. After compilation, the interface gives options for execution, including moving to a notebook, which would enable the interactive analysis described in Example 1.2.

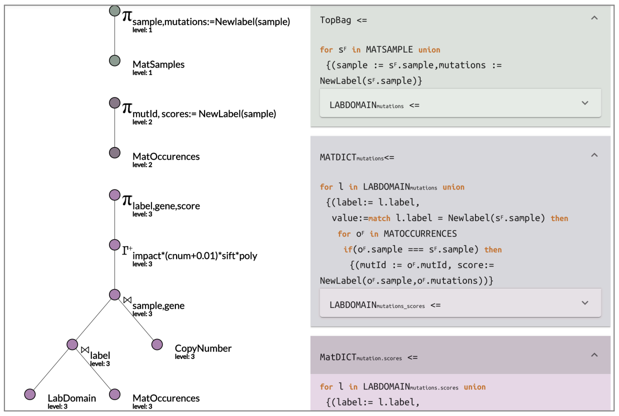
The results view (Figure 2c) displays the raw output of an execution. The user can browse through the data to drill down to a new level, and a bar graph provides an overview of a given inner collection within the output. Runtime metrics can also be explored through the interface. The distribution of partition sizes is provided, and we utilize it to give insight into skew and its management within the TraNCE architecture. This view also has links to statistics produced by the target parallel processing framework, be it Spark or Flink.

The demo offers predefined schemas and queries matching the scenarios in Section 4. Conference attendees will also be able to create schemas and queries using graphical frontends and alternative textual formats, such as SQL and JSONiq.



for in select a object union
{(
)}
select attribute
(
).groupBy({ }, { })
&&
||
:=
:=
for s in samples union
{(sample := s.bcr_patient_uuid
cands :=
for o in occurrences union
if (o.sample == s.bcr_patient_uuid)
then {(mutid := o.mutid
scores :=
(for cc in o.candidates union
for c in copyNumber union
if (cc.gene == c.
then {(gene := c.gene
score := cc.impact
)}
).groupBy({ gene }, { score })
)}

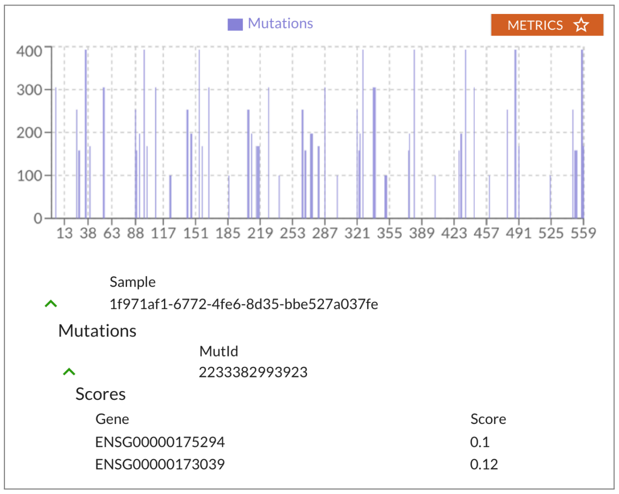
(a) Query builder view



π sample,mutations:=Newlabel(sample) level: 1
MatSamples level: 1
π mutId, scores:= NewLabel(sample) level: 2
MatOccurences level: 2
π label,gene,score level: 3
Γ^{+} impact*(cnum+0.01)*sift*poly level: 3
$\bowtie$ sample,gene level: 3
$\bowtie$ label level: 3
CopyNumber level: 3
LabDomain level: 3
MatOccurences level: 3

```
TopBag <=

for sF in MATSAMPLE union
 {(sample := sF.sample,mutations :=
NewLabel(sF.sample)}
  LABDOMAINmutations <=

MATDICTmutations<=

for l in LABDOMAINmutations union
 {(label:= l.label,
  value:=match l.label = Newlabel(sF.sample) then
   for oF in MATOCCURRENCES
    if(oF.sample === sF.sample) then
     {(mutId := oF.mutId, score:=
NewLabel(oF.sample,oF.mutations))}
  LABDOMAINmutations_scores <=

MatDICTmutation.scores <=

for l in LABDOMAINmutations.scores union
 {(label:= l.label,
```

(b) Compilation view



Sample
1f971af1-6772-4fe6-8d35-bbe527a037fe
Mutations
MutId
2233382993923
Scores

Gene	Score
ENSG00000175294	0.1
ENSG00000173039	0.12

(c) Results view

Figure 2: Some views provided by the TraNCE interface

REFERENCES

[1] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *SIGMOD*.

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

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

[5] Leonidas Fegaras and Md Hasanuzzaman Noor. 2020. Translation of Array-Based Loops to Distributed Data-Parallel Programs. In *VLDB*.

[6] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. In *VLDB*.

[7] Ingo Müller, Ghislain Fourny, Stefan Irimescu, Can Berker Cikis, and Gustavo Alonso. 2021. Rumble: Data Independence for Large Messy Data Sets. In *VLDB*.

[8] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. 2017. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B B)*. 21–24. https://doi.org/10.1109/BLOCKS.2017.8120404

[9] Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. 2021. Scalable Querying of Nested Data. In *VLDB*.

[10] Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Yao Shi. 2020. Scalable Analysis of Multi-Modal Biomedical Data. bioarxiv.org.

[11] Alexander Ulrich. 2019. *Query Flattening and the Nested Data Parallelism Paradigm*. Ph.D. Dissertation. University of Tübingen, Germany. https://publikationen.uni-tuebingen.de/xmlui/handle/10900/87698/

[12] Alexander Ulrich and Torsten Grust. 2015. The Flatter, the Better: Query Compilation Based on the Flattening Transformation. In *SIGMOD*.