Optimization of Recommendation Pipelines using Apache Spark

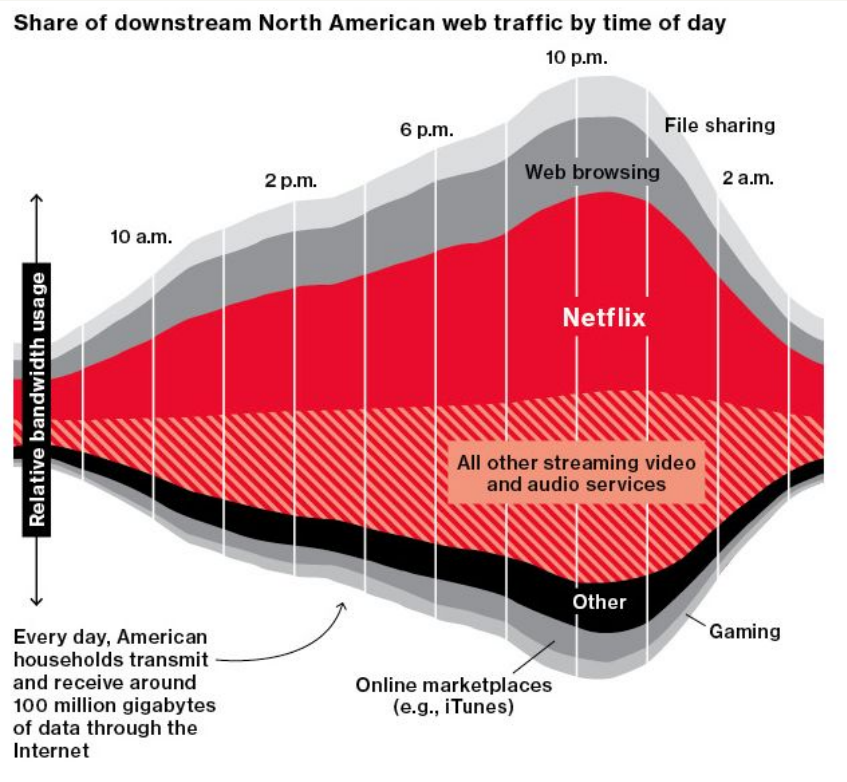
Hua Jiang and DB Tsai

Spark Summit SF - June 6, 2017

NETFLIX

Netflix Scale

- Started streaming videos 10 years ago
- > 100M members
- > 190 countries
- > 1000 device types
- A third of peak US downstream traffic



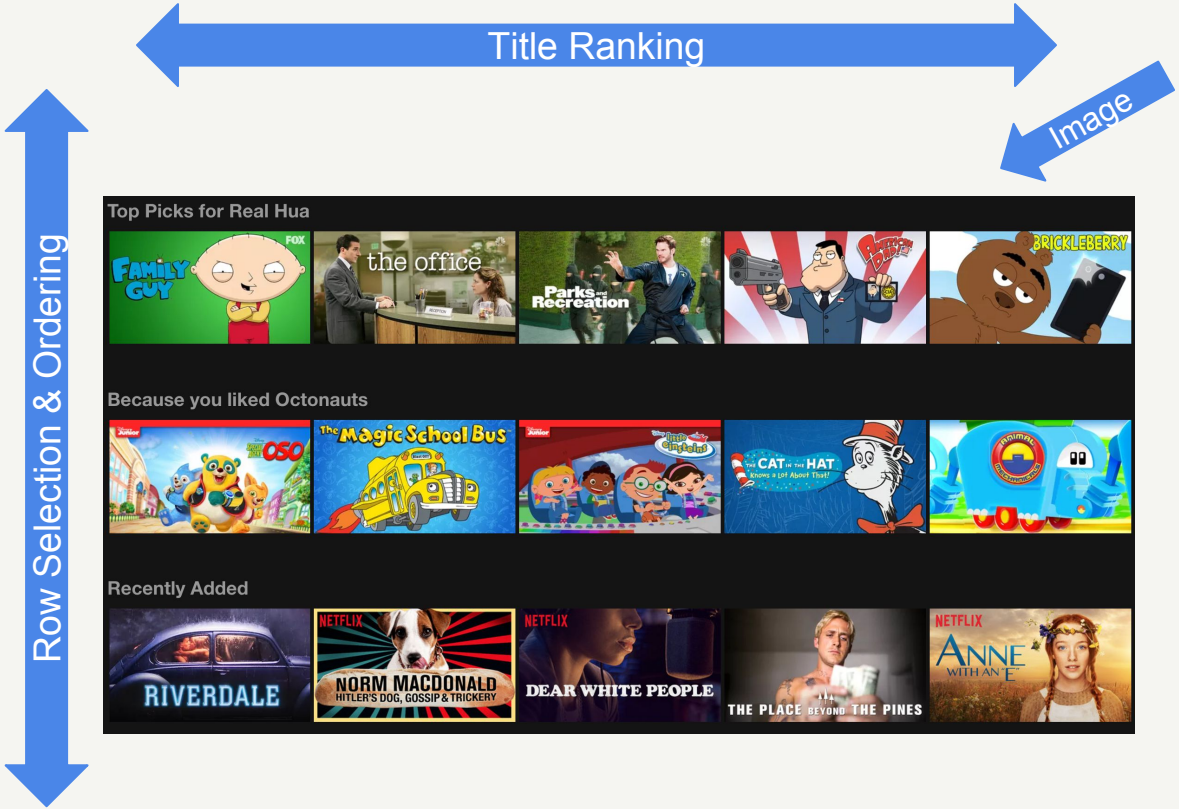
Recommendation System: Ideal State

Turn on Netflix, and the

absolute best content for you

would **automatically start** playing

Everything is a Recommendation



Recommendations are driven by machine learning algorithms

Over 80% of what members watch comes from our recommendations

Running Experiments

- Try an **idea offline using historical data** to see if it would have made better recommendations

- If it would, deploy a live **A/B test** to see if it performs well in production

Running Experiments

Offline Experiment

Online System

Design a New Experiment to Test Out Different Ideas

Design Experiment

Collect Label Dataset

Offline Feature Generation

Model Training

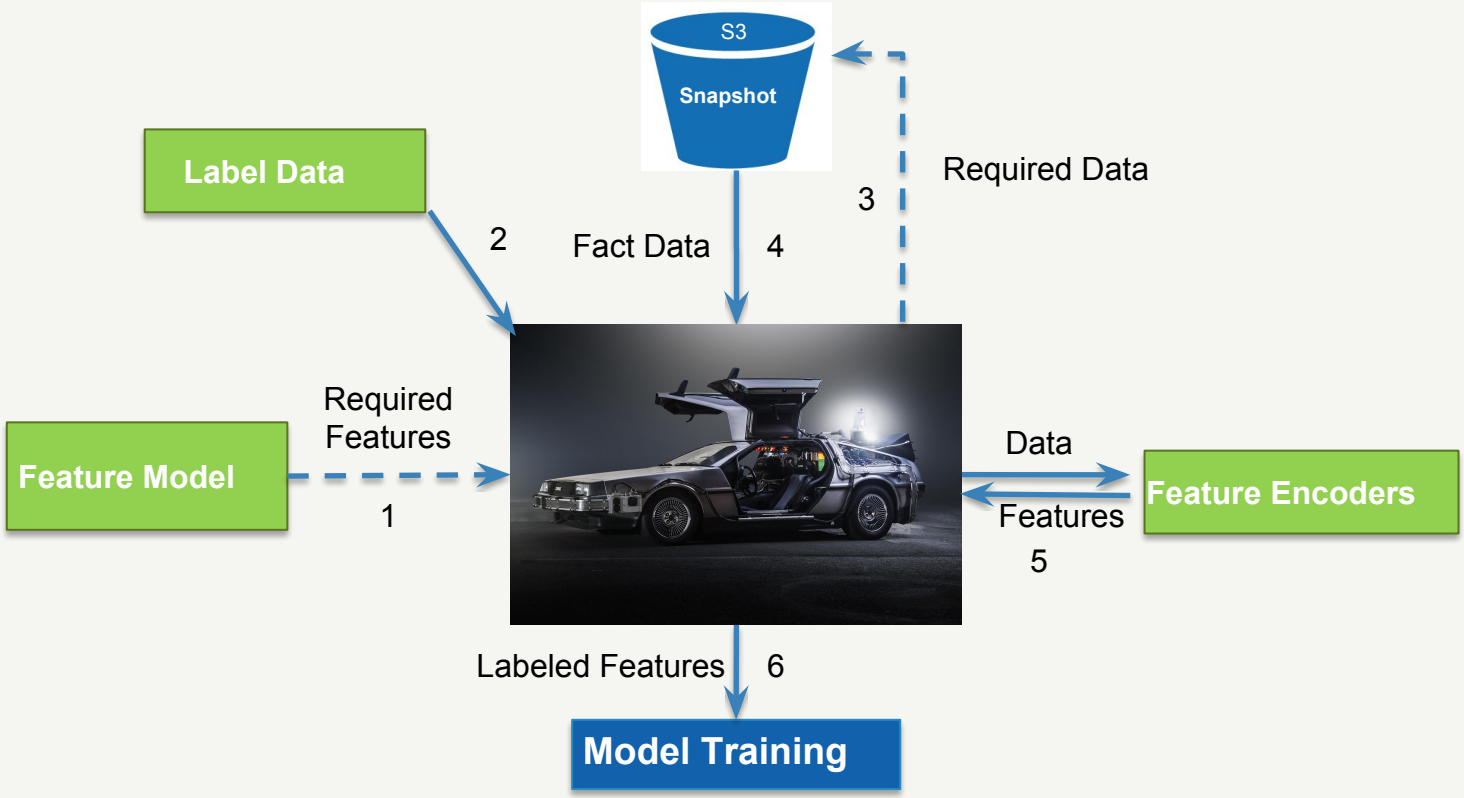
Compute Validation Metrics

Model Testing

Online AB Testing

Spark

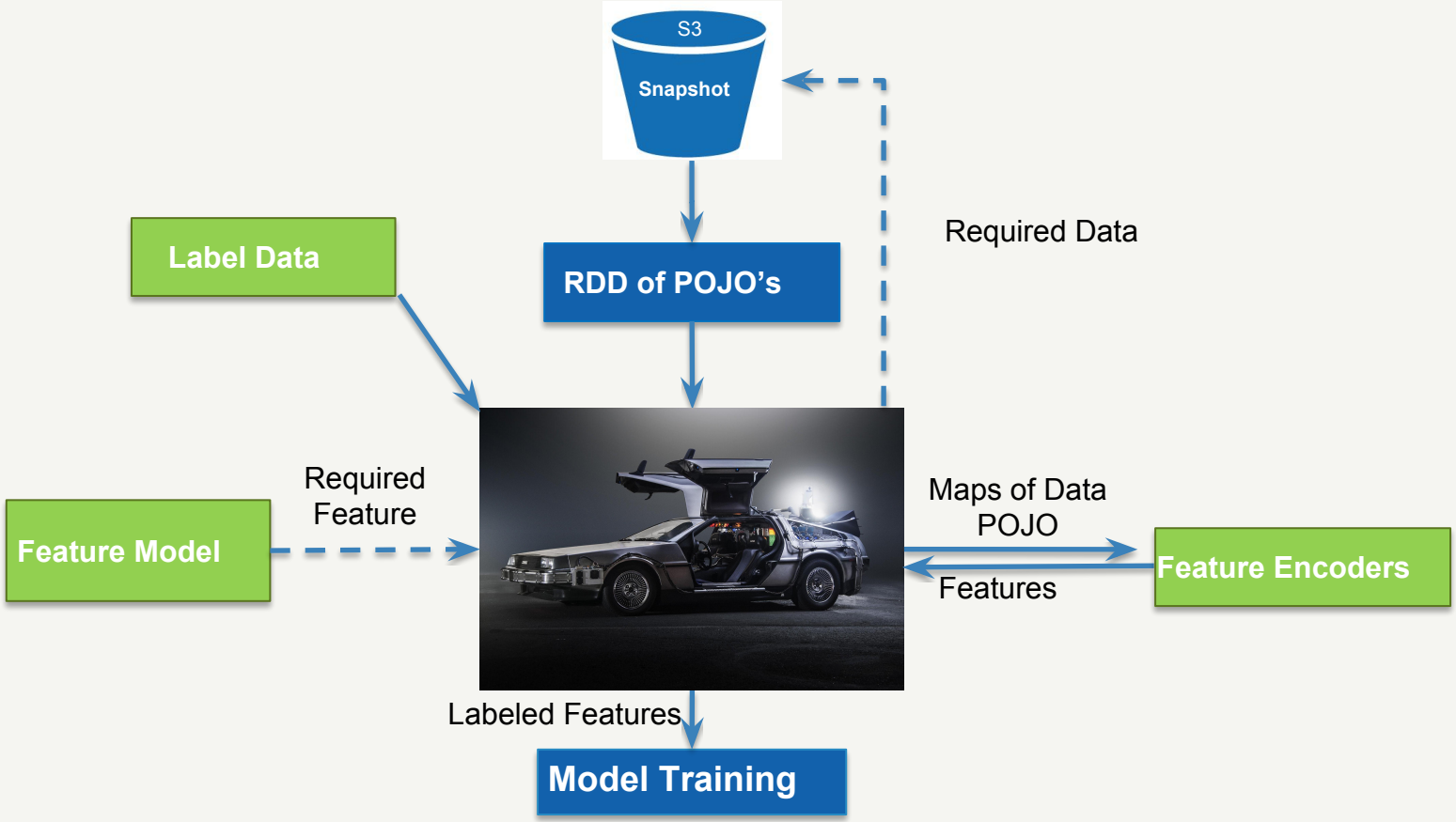
Feature Generation: Feature Computation



Version 1: RDD-Based Feature Generation

- RDD: Resilient Distributed Dataset
- Our first version was written when only RDD operations were available
- Opacity
  - Data are opaque
  - Computation is opaque

Version 1: RDD-Based Feature Generation



Version 1: RDD-Based Feature Generation

RDD operations are at low level.

You are responsible for performance optimization.

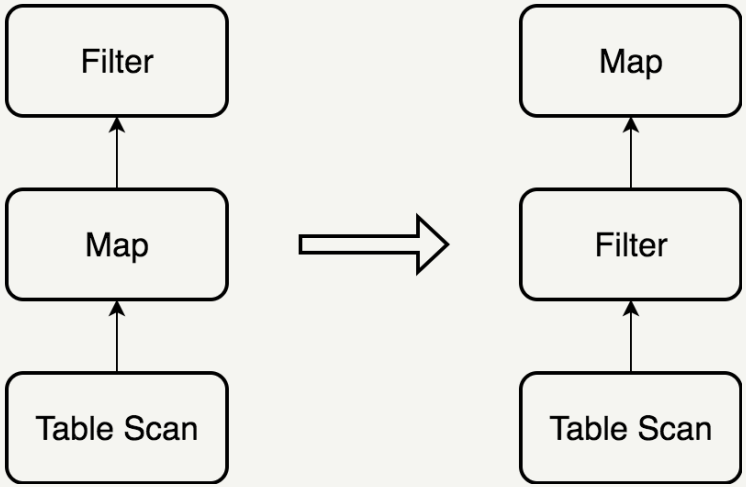
RDD operations are on whole objects, even if only one field is required.

Version 2: Using DataFrame

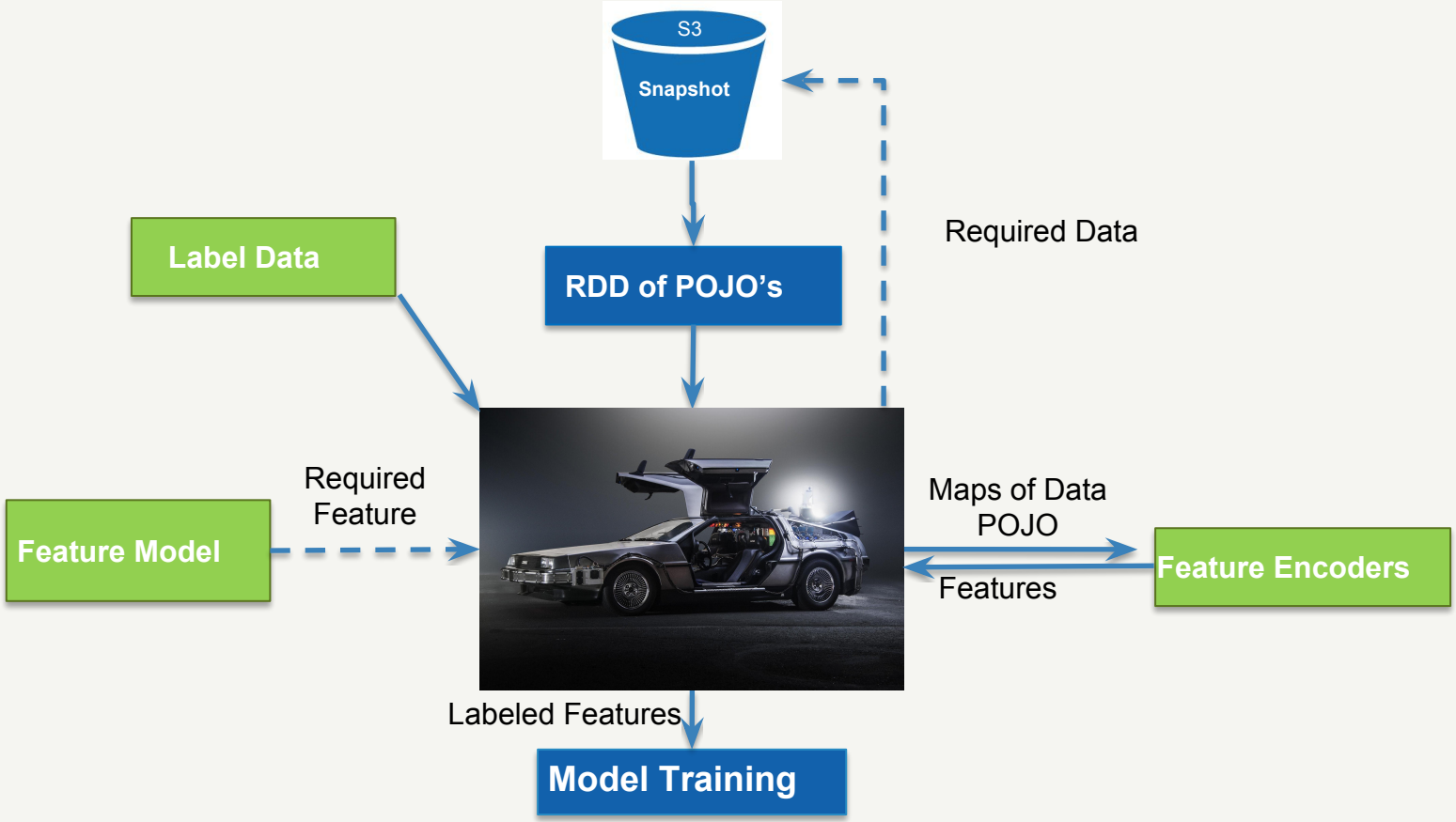
- DataFrame: Structured Data Organized into Named Columns
- Transparency
  - Data are structured
  - Computations are planned based on common patterns

Version 2: Using DataFrame

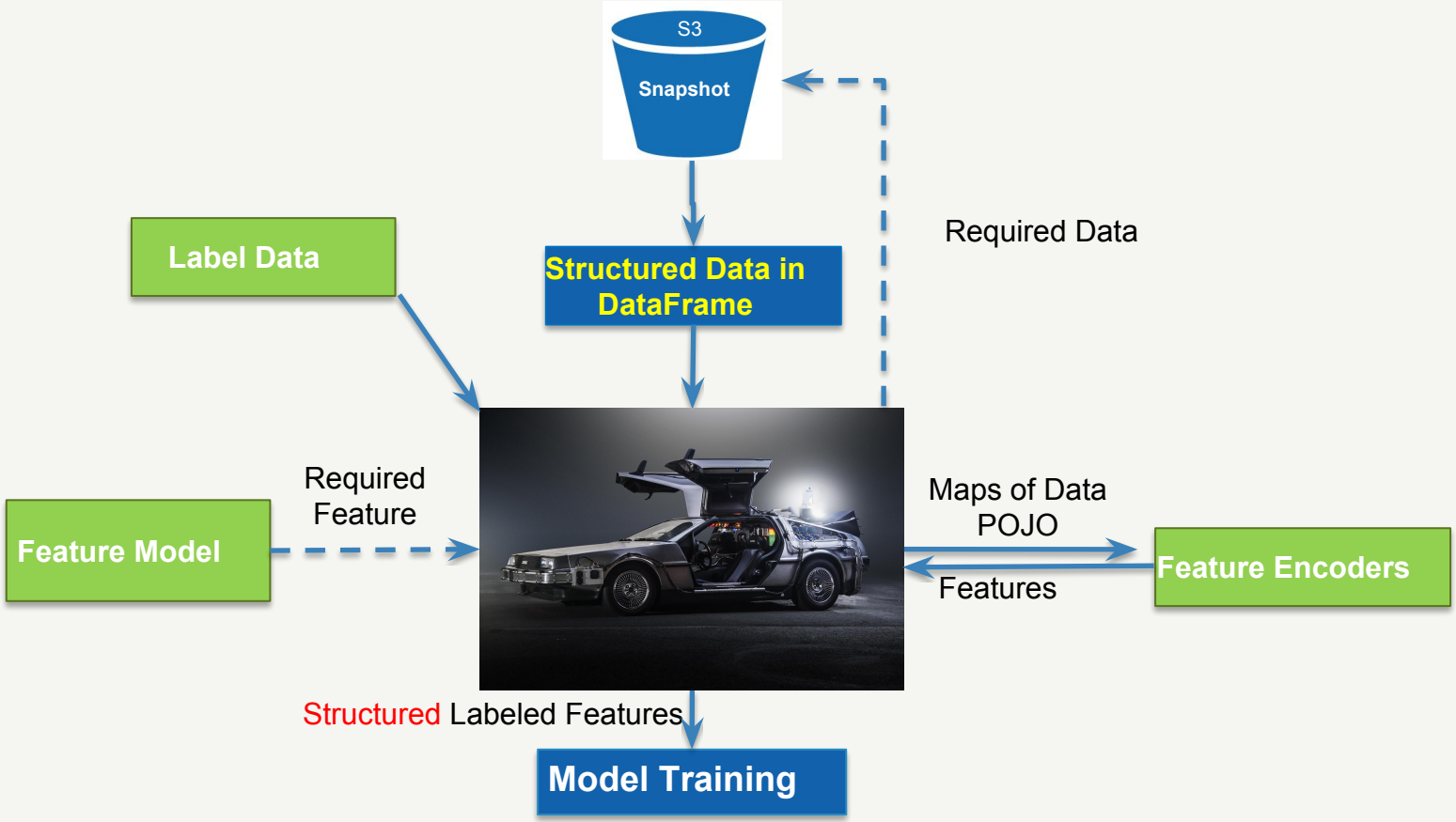
Spark SQL optimizer, Catalyst, optimizes DataFrame operation



Version 2: Using DataFrame

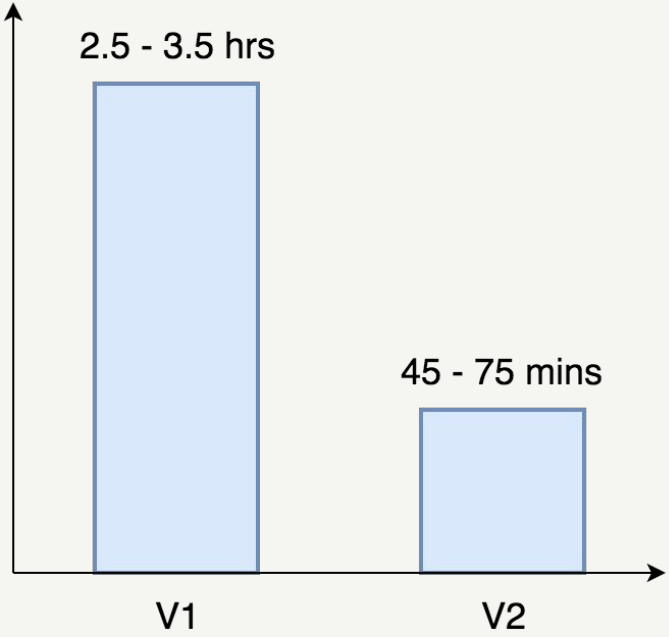


Version 2: Using DataFrame



Version 2: Using DataFrame

- 50 ~ 80 executors
- ~3 cores per executor
- ~24GB per executor



~3x run time gain in feature generation

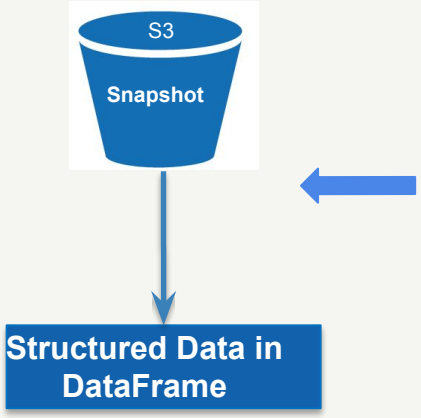
Version 2: Using DataFrame

Let’s take a look at the physical plan of the DataFrame taken from snapshot...

```
== Physical Plan ==
Project [...]
+- Filter (...)
   +- Scan ExistingRDD[...]
```

Version 2: Using DataFrame (with RDD[Row])

We use RDD[Row] from data frame and create a new data frame by manipulating the Row object.



Deduping Logic with Row Manipulation

Version 2: Using DataFrame (with RDD[Row])

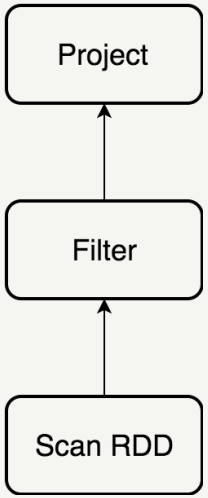
Even the new DataFrame, created from RDD[Row], has columns with the same names, they are different to Spark

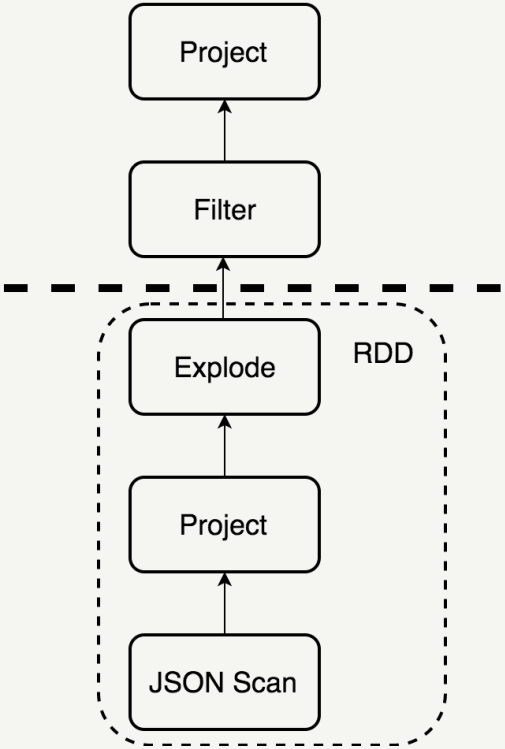
col1#5	col2#6	col3#7

col1#12	col2#13	col3#14

Version 2: Using DataFrame (with RDD[Row])

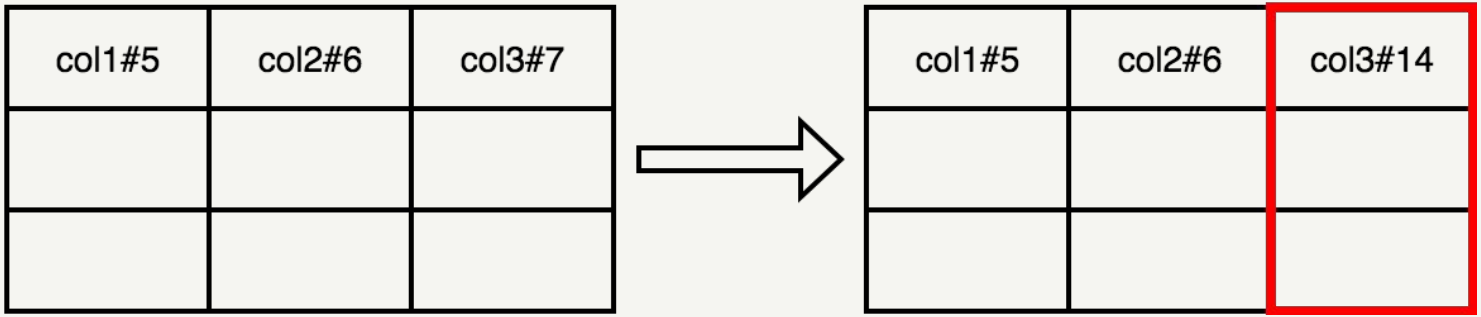
Manipulations on row objects are completely opaque, blocking optimizer from moving operations around.





Version 3: Column Operations To The Rescue

Most of the operations are essentially column(s) to column(s)



Version 3: Column Operations To The Rescue

Most of the operations are essentially column(s) to column(s)

Possible Replacement for row manipulations:

- Spark SQL Functions
- User-Defined Functions
- Catalyst Expression

Version 3: Column Operations To The Rescue

Spark SQL Functions
(org.apache.spark.sql.functions)

- Built-in
- Highly efficient
  - Internal data structure
  - Code generation
  - Supports rule-based optimization
- A variety of categories
  - Aggregation
  - Collection
  - Math
  - String

Version 3: Column Operations To The Rescue

User-Defined Functions (UDFs)

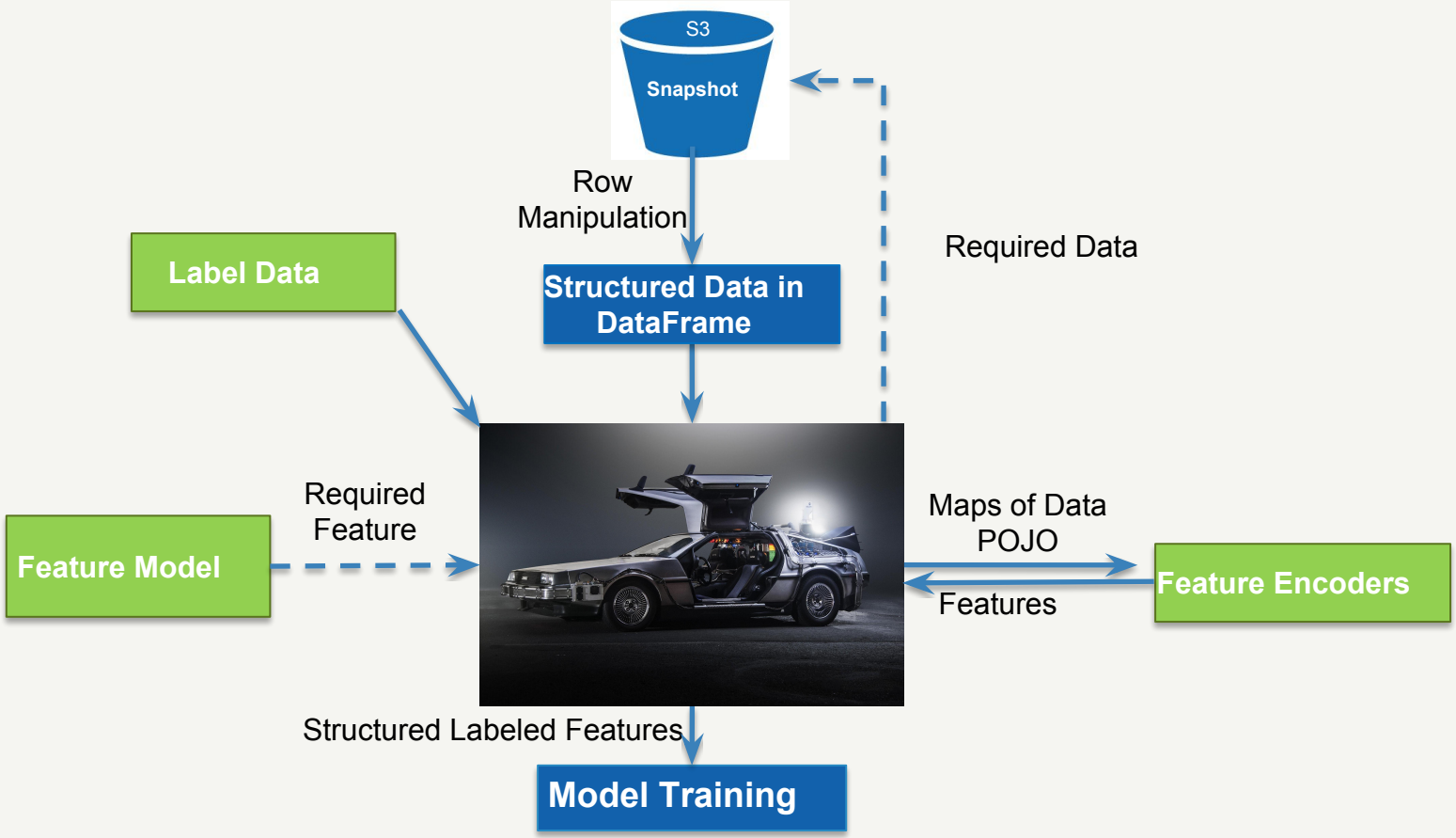
- Scala functions with certain types
- Highly flexible
- Data encoding/decoding required

Version 3: Column Operations To The Rescue

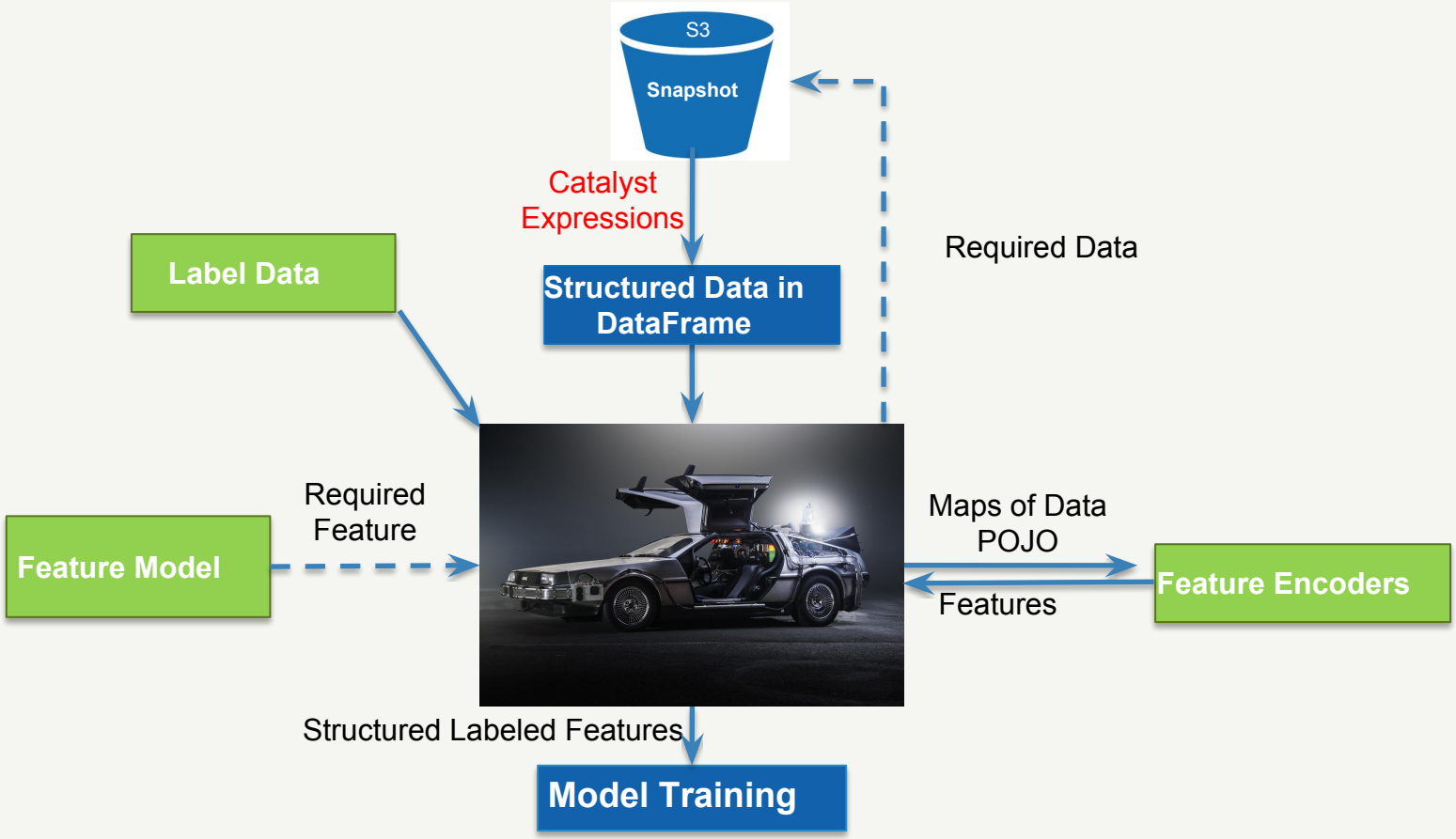
User-Defined Catalyst Expressions

- Flexible
  - User defines the operations
- Efficient
  - Internal data structure
  - Code generation possible

Version 3: Column Operations To The Rescue

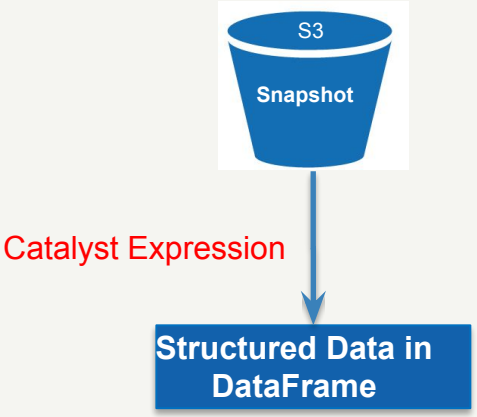


Version 3: Column Operations To The Rescue



Version 3: Column Operations To The Rescue

We replaced row manipulation with Catalyst expression

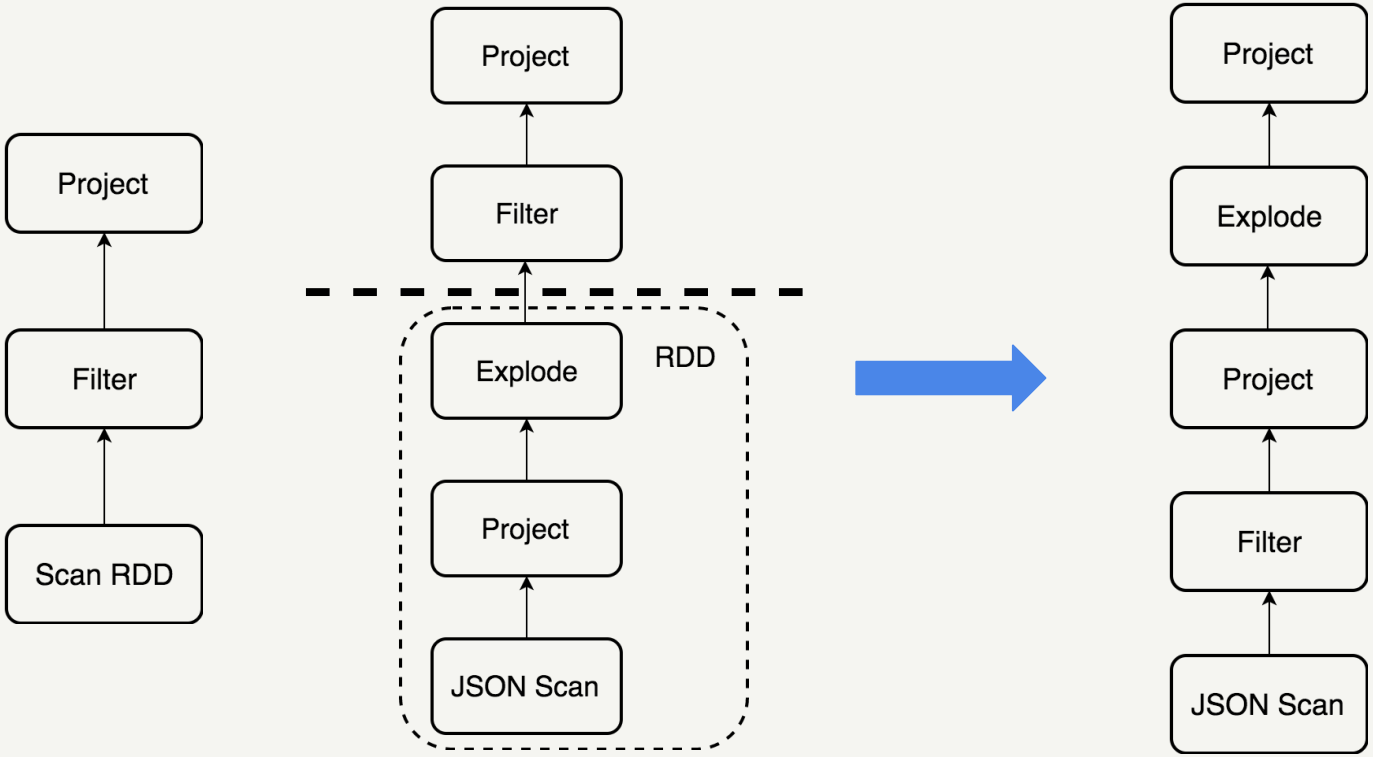



```
case class RemoveDuplications(child: Expression) extends
UnaryExpression {

 ...

}
```

Version 3: Column Operations To The Rescue



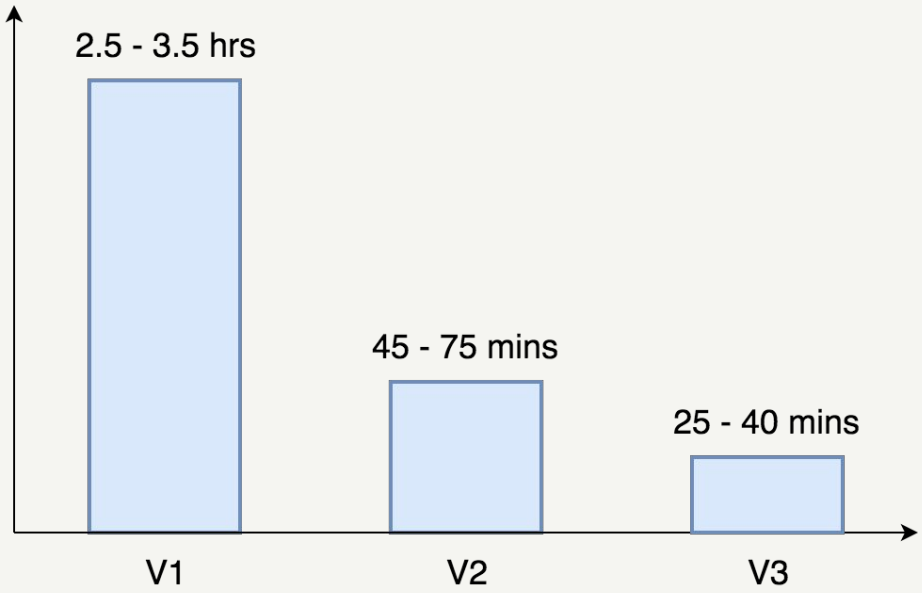
Version 3: Column Operations To The Rescue

Physical Plan with Column Operations

```
== Physical Plan ==
Project [...]
+- BroadcastHashJoin (...)
   :- *Filter (...)
   :  +- LocalTableScan [...]
   +- BroadcastExchange HashedRelationBroadcastMode(...)
      +- Project [...]
         +- Generate explode(...), true, false, [...]
            +- Project [...]
               +- Filter (...)
                  +- Scan json (...)
```

Version 3: Column Operations To The Rescue

- 50 ~ 80 executors
- ~3 cores per executor
- ~24GB per executor



~2x run time gain compared to version 2

Conclusions

- Time Travel in Offline Training
  - Fact logging + offline feature generation
- Optimization
  - Remove “black boxes”
  - Prefer high-level DataFrame APIs
  - Prefer column operations over row manipulations

Questions?

(We are hiring...)