SPARK+AI
SUMMIT 2018

Apache Spark for library developers

William Benton
willb@redhat.com
@willb

Erik Erlandson
eje@redhat.com
@manyangled

The Silex and Isarn libraries

Reusable open-source code that works with Spark, factored from internal apps.

We've tracked Spark releases **since Spark 1.3.0**.

See **https://silex.radanalytics.io** and **http://isarnproject.org**

DANGER
NO DIVING

Forecast

Basic considerations for reusable Spark code

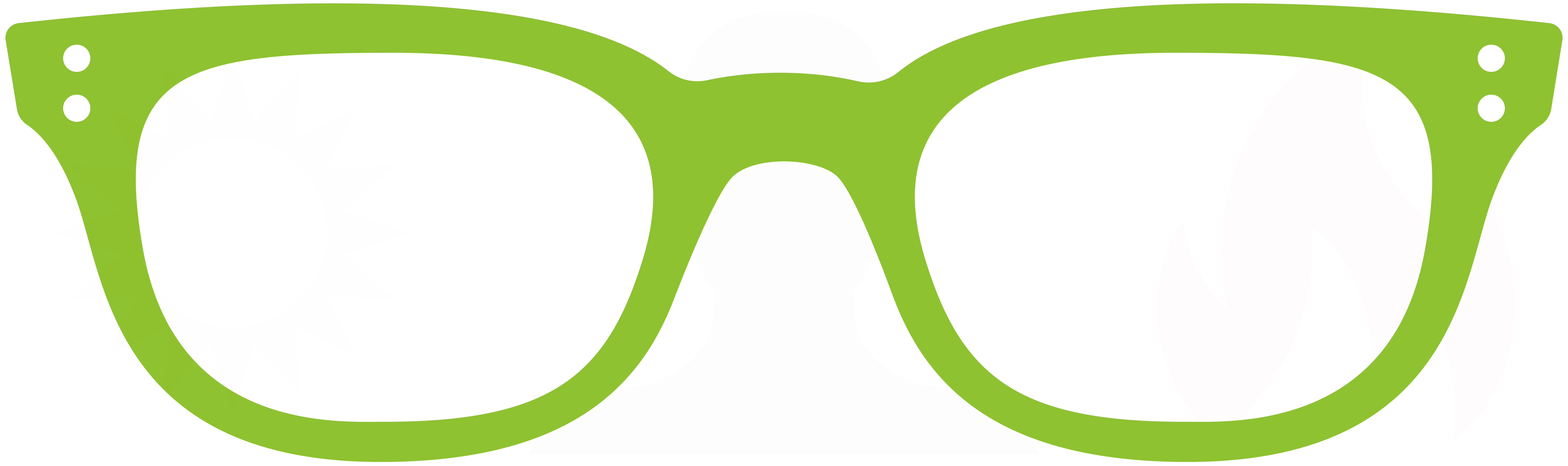
Generic functions for parallel collections

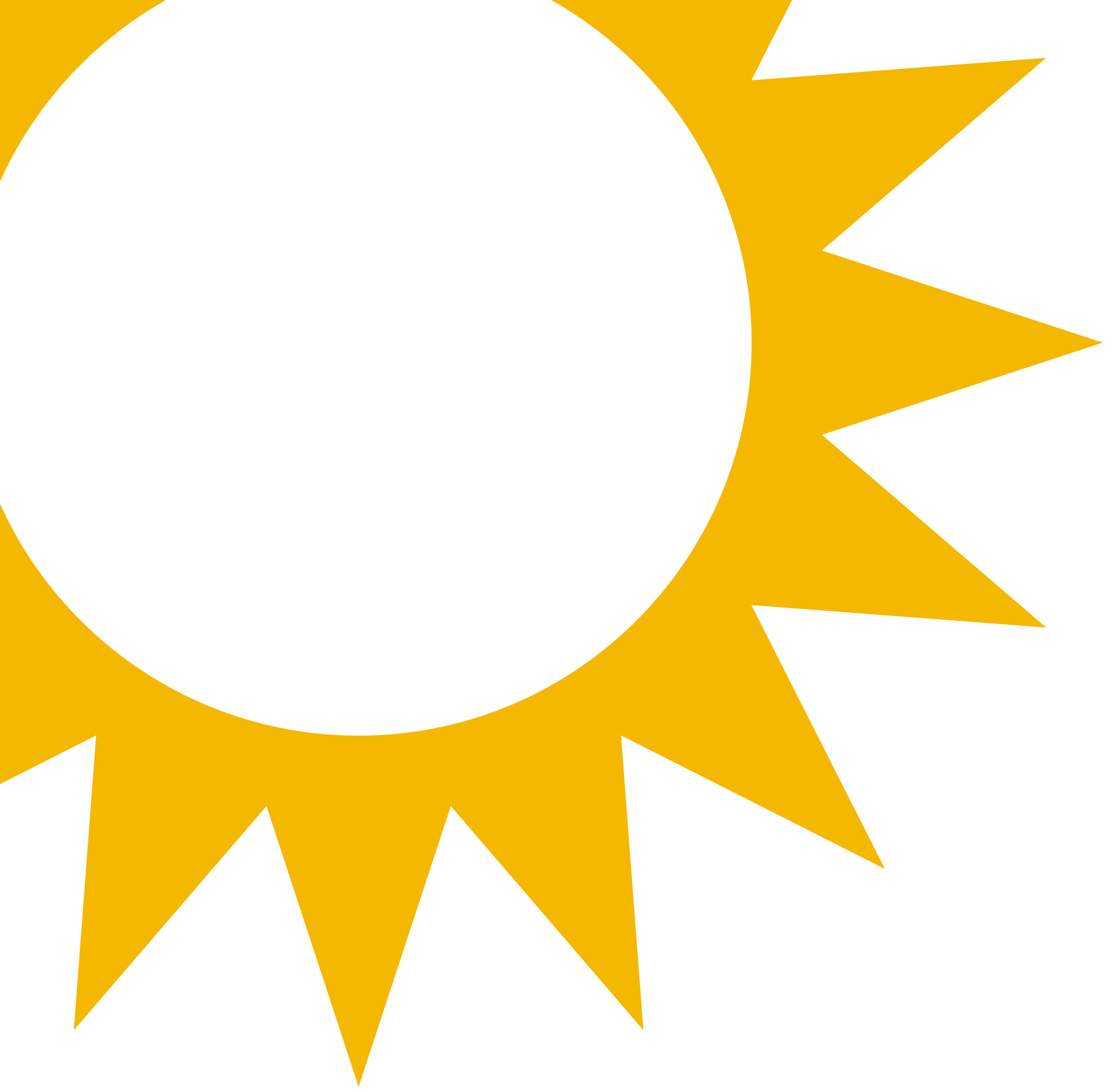
Extending data frames with custom aggregates

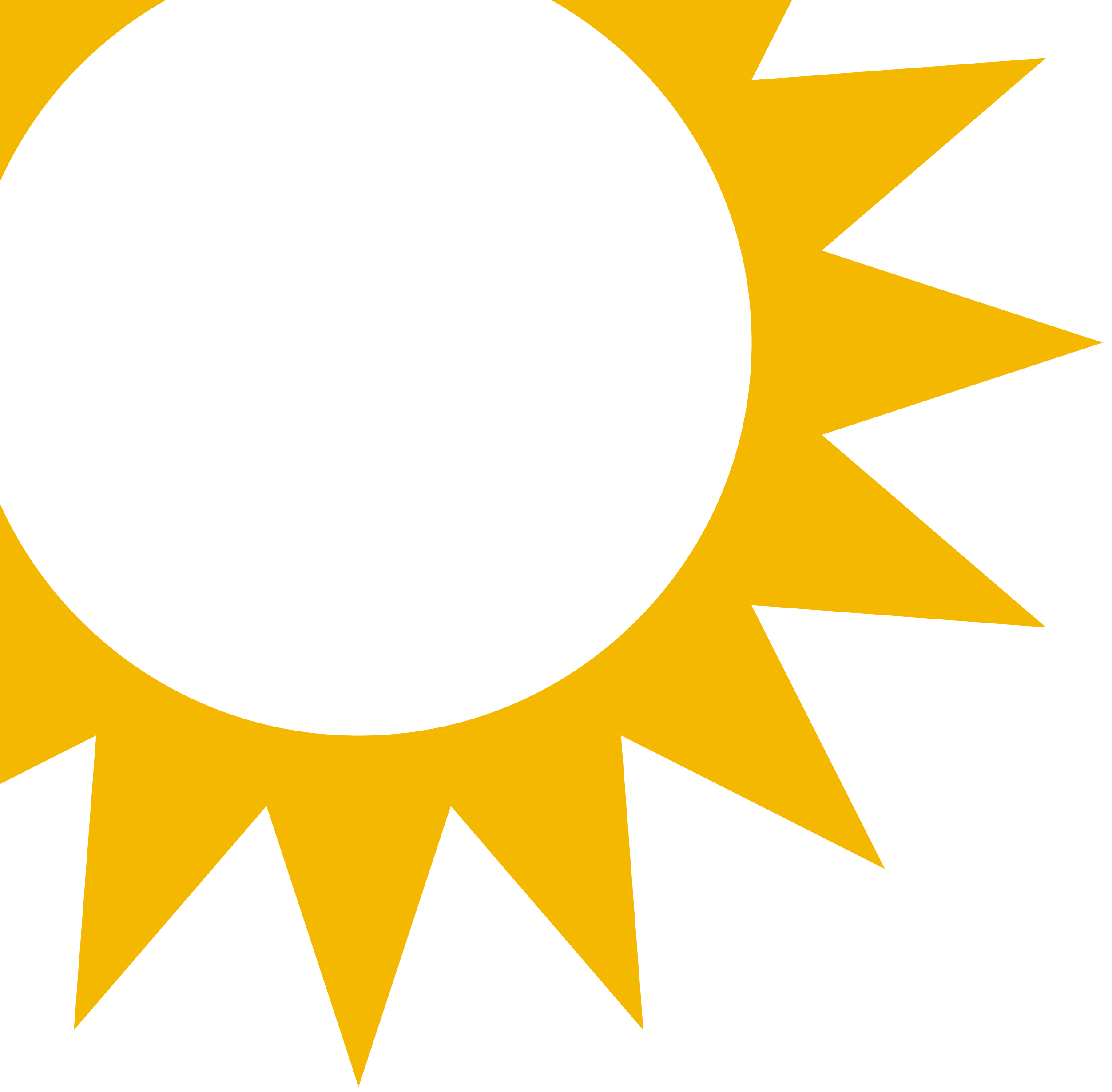
Exposing JVM libraries to Python

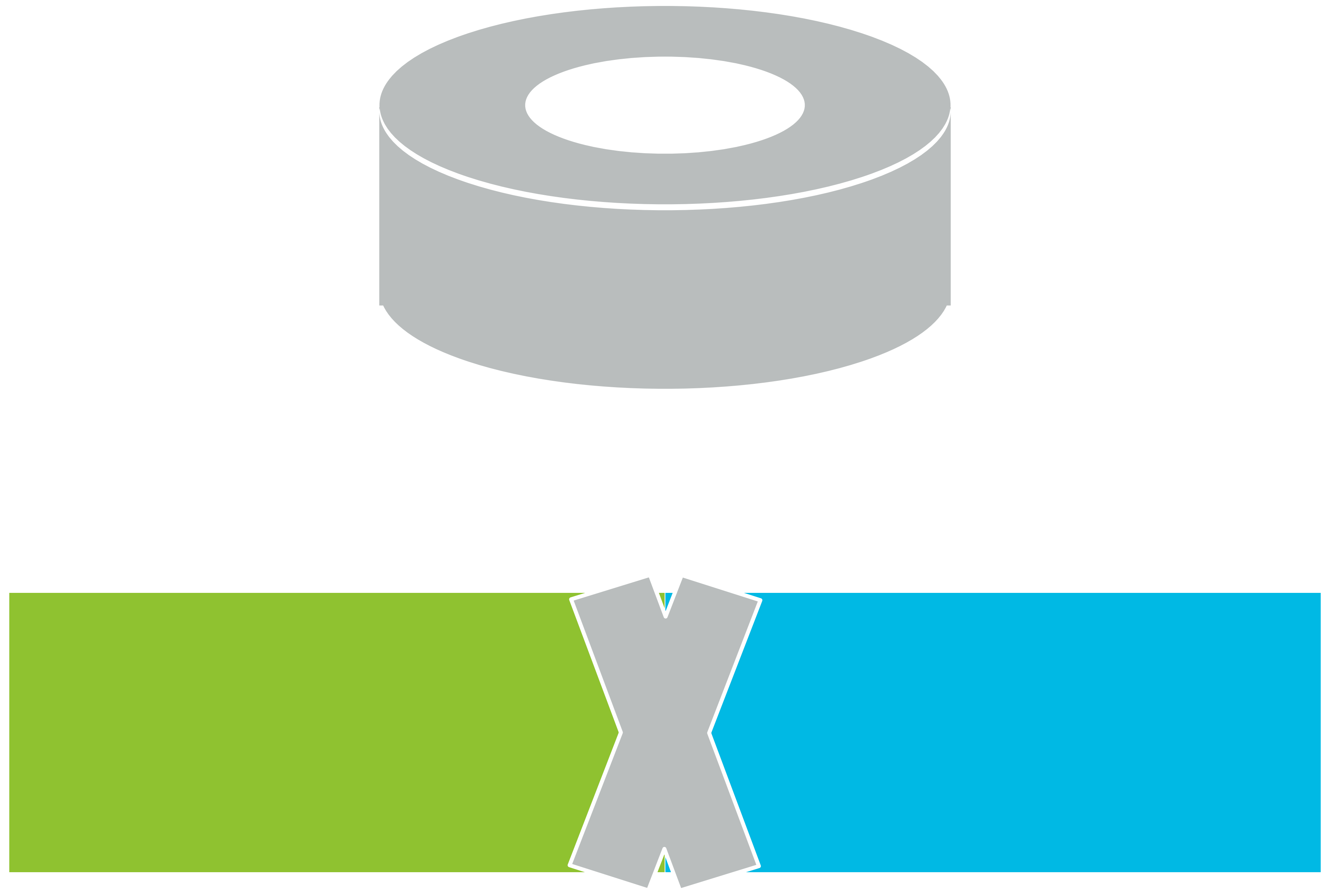
Sharing your work with the world

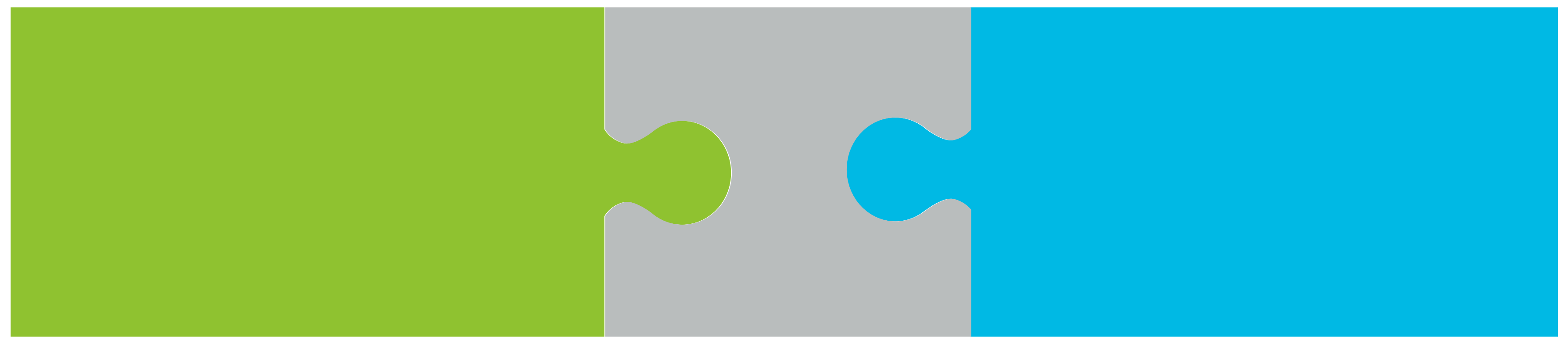
Basic considerations



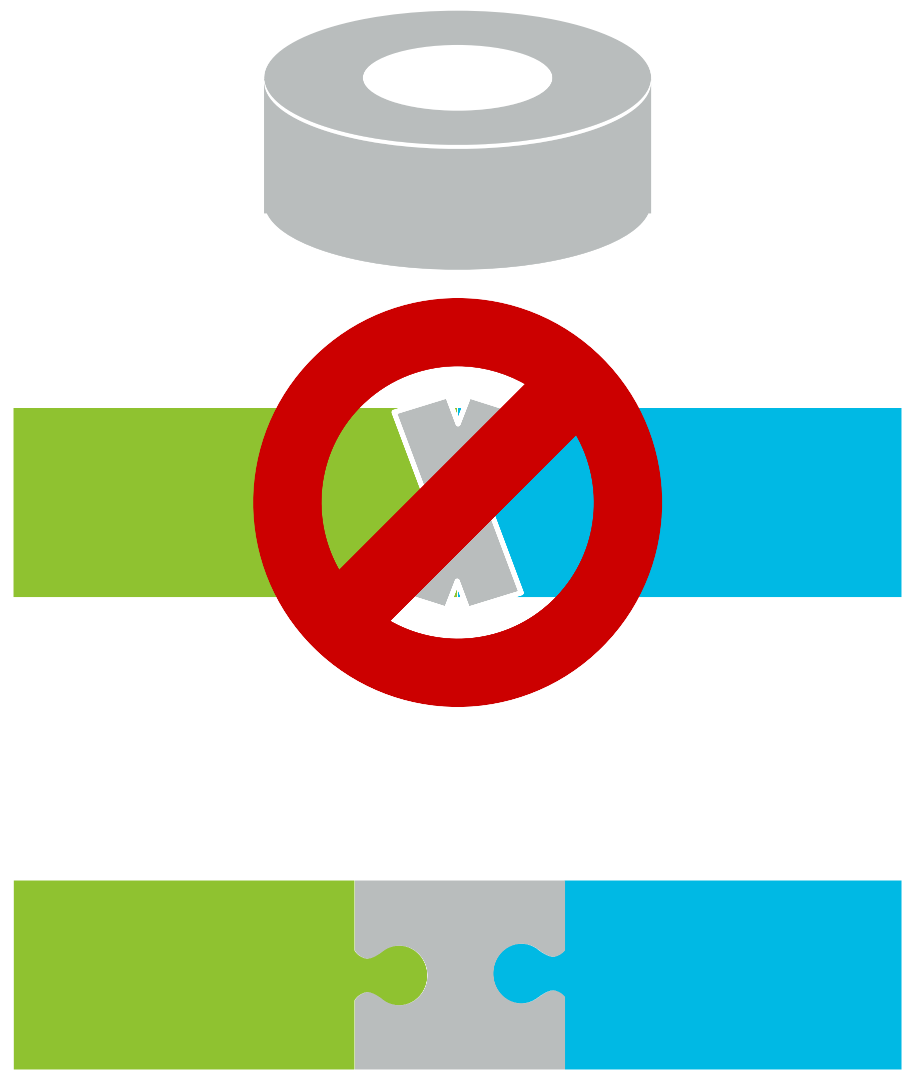








Today’s main themes



Cross-building for Scala

in your SBT build definition:

```
scalaVersion := "2.11.11"

crossScalaVersions := Seq("2.10.6", "2.11.11")
```

in your shell:

```
$ sbt +compile
$ sbt "++ 2.11.11" compile
```

Cross-building for Scala

in your SBT build definition:

```
scalaVersion := "2.11.11"

crossScalaVersions := Seq("2.10.6", "2.11.11")
```

in your shell:

```
$ sbt +compile          # or test, package, publish, etc.
$ sbt "++ 2.11.11" compile
```

Cross-building for Scala

in your SBT build definition:

```
scalaVersion := "2.11.11"

crossScalaVersions := Seq("2.10.6", "2.11.11")
```

in your shell:

```
$ sbt +compile          # or test, package, publish, etc.
$ sbt "++ 2.11.11" compile
```

“Bring-your-own Spark”

in your SBT build definition:

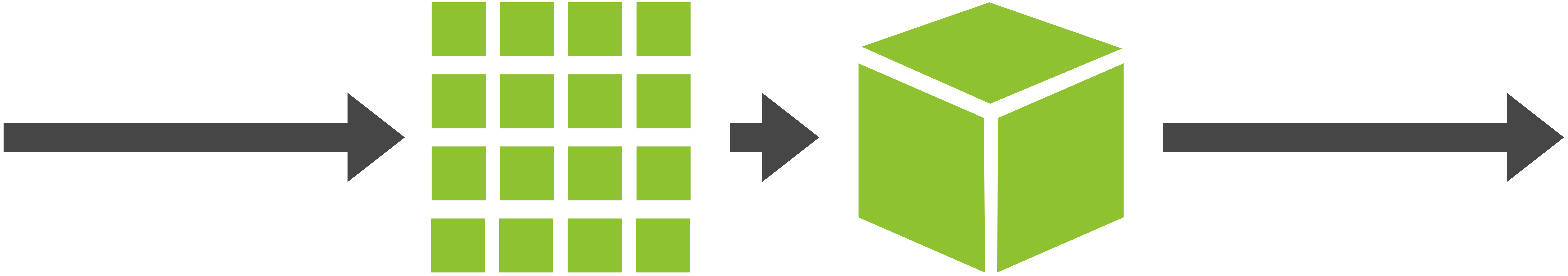
```
libraryDependencies ++= Seq(
    "org.apache.spark" %% "spark-core" % "2.3.0" % Provided,
    "org.apache.spark" %% "spark-sql" % "2.3.0" % Provided,
    "org.apache.spark" %% "spark-mllib" % "2.3.0" % Provided,
    "joda-time" % "joda-time" % "2.7",
    "org.scalatest" %% "scalatest" % "2.2.4" % Test)
```

“Bring-your-own Spark”

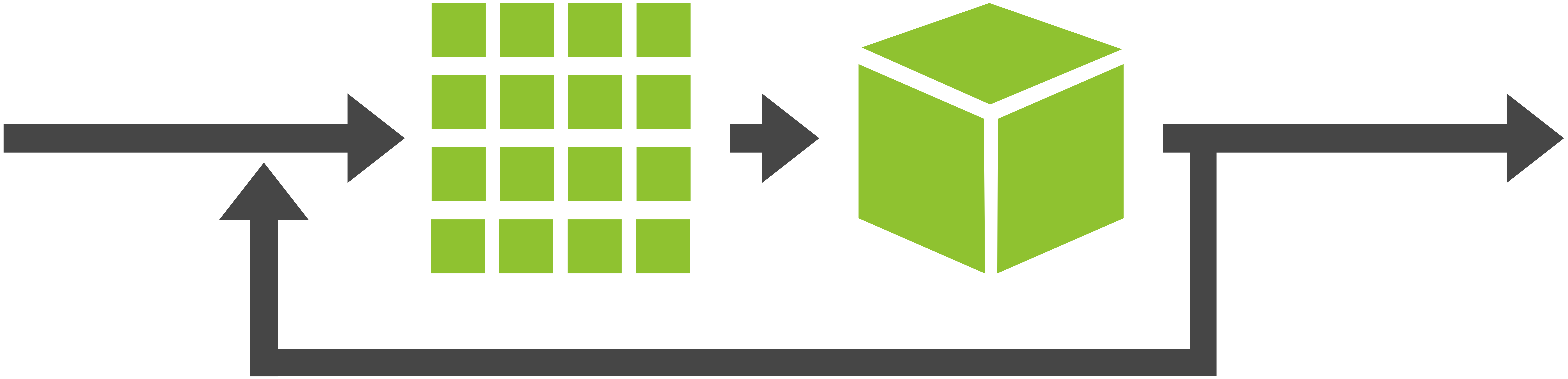
in your SBT build definition:

```
libraryDependencies ++= Seq(
    "org.apache.spark" %% "spark-core" % "2.3.0" % Provided,
    "org.apache.spark" %% "spark-sql" % "2.3.0" % Provided,
    "org.apache.spark" %% "spark-mllib" % "2.3.0" % Provided,
    "joda-time" % "joda-time" % "2.7",
    "org.scalatest" %% "scalatest" % "2.2.4" % Test)
```

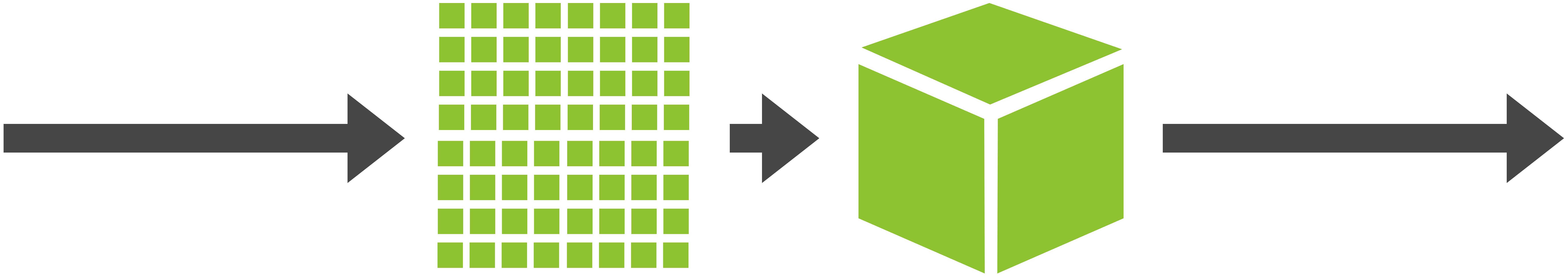
Taking care with resources



Taking care with resources



Taking care with resources



Caching when necessary

```
def step(rdd: RDD[_]) = {

  rdd.cache()
  result = trainModel(rdd)


  result
}
```

Caching when necessary

```
def step(rdd: RDD[_]) = {

  rdd.cache()
  result = trainModel(rdd)


  result
}
```

Caching when necessary

```
def step(rdd: RDD[_]) = {

  rdd.cache()
  result = trainModel(rdd)

  rdd.unpersist()

  result
}
```

Caching when necessary

```
def step(rdd: RDD[_]) = {
  val wasUncached = rdd.storageLevel == StorageLevel.NONE
  if (wasUncached) { rdd.cache() }
  result = trainModel(rdd)



  result
}
```

Caching when necessary

```
def step(rdd: RDD[_]) = {
  val wasUncached = rdd.storageLevel == StorageLevel.NONE
  if (wasUncached) { rdd.cache() }
  result = trainModel(rdd)
  if (wasUncached) { rdd.unpersist() }

  result
}
```

```
var nextModel = initialModel
for (int i = 0; i < iterations; i++) {
  val current = sc.broadcast(nextModel)
  val newState = examples.aggregate(ModelState.empty()) (
    { case (state: ModelState, example: Example) =>
      state.update(current.value.lookup(example, i), example) }
    { case (s1: ModelState, s2: ModelState) => s1.combine(s2) }
  )
  nextModel = modelFromState(newState)
  current.unpersist
}
```

```
var nextModel = initialModel
for (int i = 0; i < iterations; i++) {
  val current = sc.broadcast(nextModel)
  val newState = examples.aggregate(ModelState.empty()) (
    { case (state: ModelState, example: Example) =>
      state.update(current.value.lookup(example, i), example) }
    { case (s1: ModelState, s2: ModelState) => s1.combine(s2) }
  )

  nextModel = modelFromState(newState)
  current.unpersist
}
```

```
var nextModel = initialModel
for (int i = 0; i < iterations; i++) {
  val current = sc.broadcast(nextModel)
  val newState = examples.aggregate(ModelState.empty()) (
    { case (state: ModelState, example: Example) =>
      state.update(current.value.lookup(example, i), example) }
    { case (s1: ModelState, s2: ModelState) => s1.combine(s2) }
  )
  nextModel = modelFromState(newState)
  current.unpersist
}
```

Writing generic code for Spark's parallel collections

The RDD is invariant

$$T <: U \not\vdash \texttt{RDD[T]} <: \texttt{RDD[U]}$$

$T <: U \not\vdash \mathrm{RDD}[T] <: \mathrm{RDD}[U]$

```scala
trait HasUserId { val userid: Int }
case class Transaction(override val userid: Int,
                       timestamp: Int,
                       amount: Double)
  extends HasUserId {}

def badKeyByUserId(r: RDD[HasUserId]) = r.map(x => (x.userid, x))
```

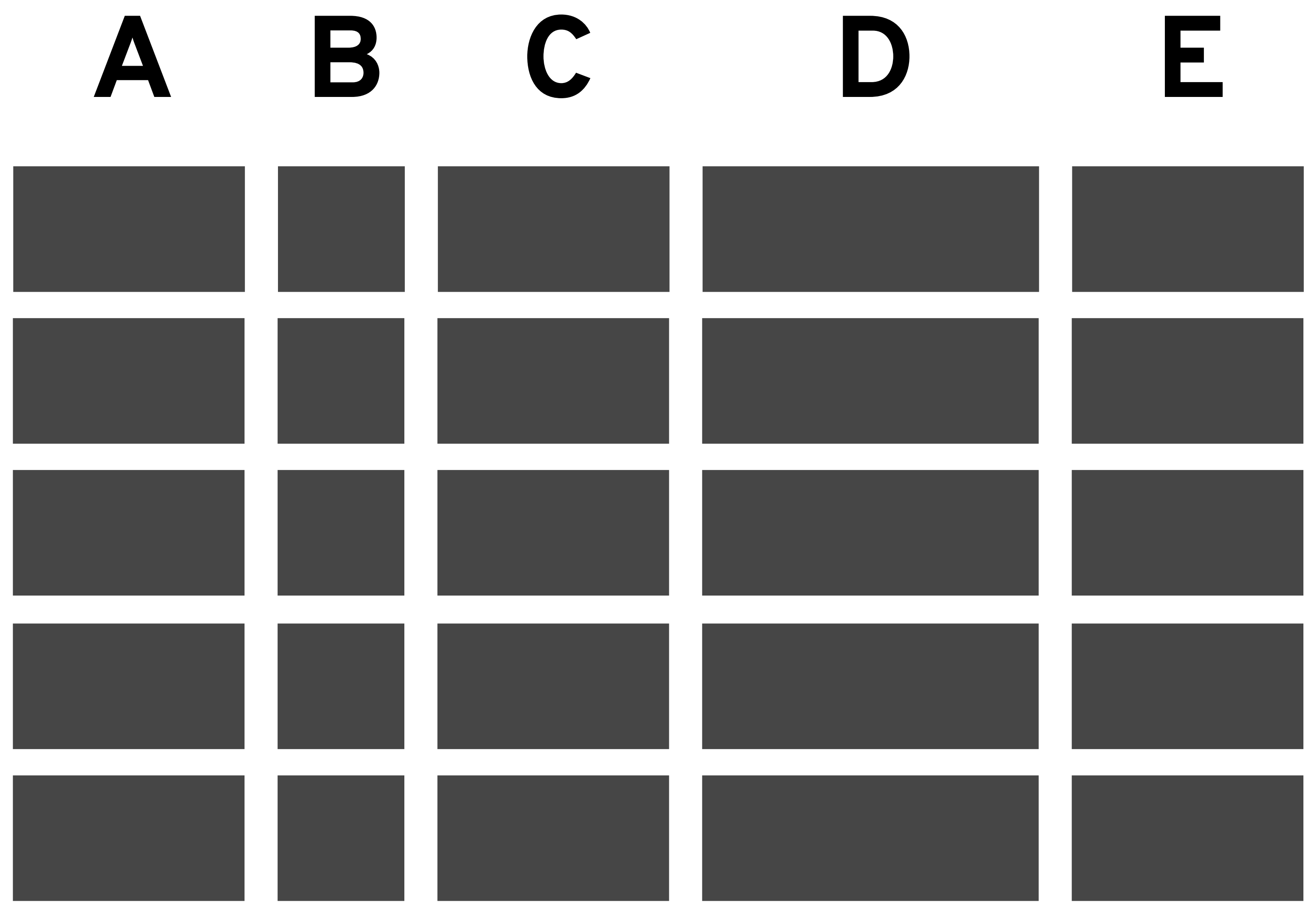
$\texttt{T <: U} \nvdash \texttt{RDD[T] <: RDD[U]}$

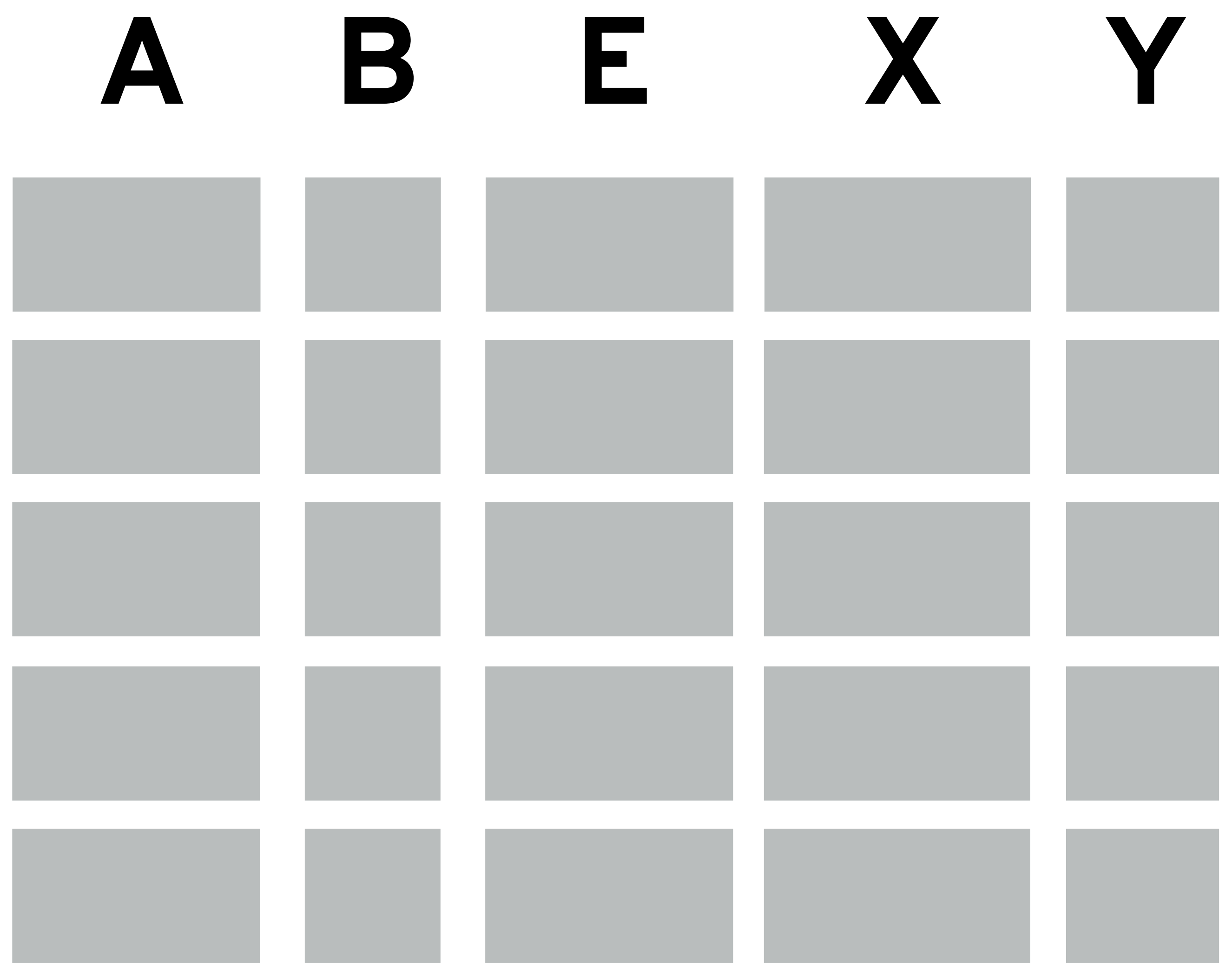
```scala
trait HasUserId { val userid: Int }
case class Transaction(override val userid: Int,
                       timestamp: Int,
                       amount: Double)
  extends HasUserId {}

def badKeyByUserId(r: RDD[HasUserId]) = r.map(x => (x.userid, x))
```

```
val xacts = spark.parallelize(Array(
  Transaction(1, 1, 1.0),
  Transaction(2, 2, 1.0)
))

badKeyByUserId(xacts)
<console>: error: type mismatch;
 found   : org.apache.spark.rdd.RDD[Transaction]
 required: org.apache.spark.rdd.RDD[HasUserId]
Note: Transaction <: HasUserID, but class RDD is invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
       badKeyByUserId(xacts)
```

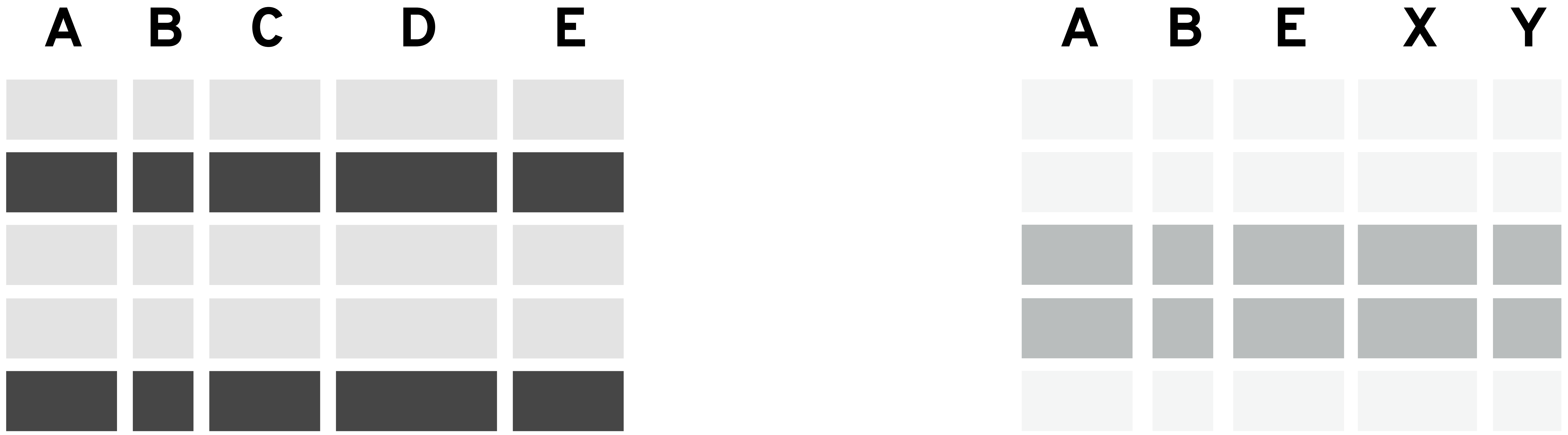
```
val xacts = spark.parallelize(Array(
  Transaction(1, 1, 1.0),
  Transaction(2, 2, 1.0)
))

badKeyByUserId(xacts)
<console>: error: type mismatch;
 found   : org.apache.spark.rdd.RDD[Transaction]
 required: org.apache.spark.rdd.RDD[HasUserId]
Note: Transaction <: HasUserID, but class RDD is invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
       badKeyByUserId(xacts)
```

An example: natural join

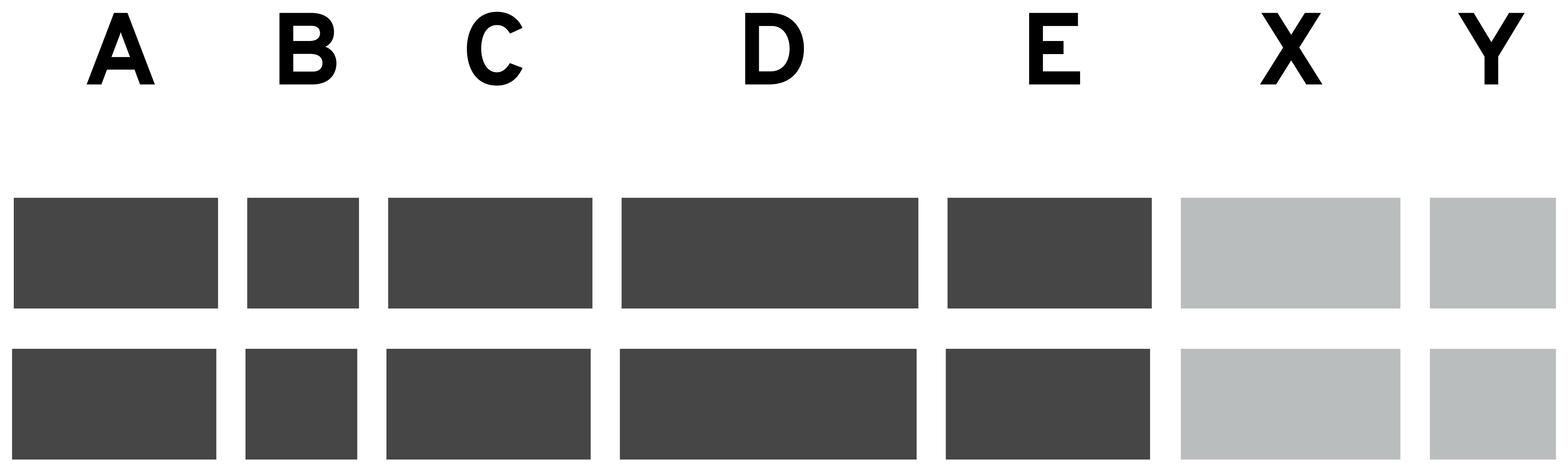




An example: natural join



An example: natural join



Ad-hoc natural join

```
df1.join(df2, df1("a") === df2("a") &&
              df1("b") === df2("b") &&
              df1("e") === df2("e"))
```

```scala
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

```scala
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

```
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

introspecting over column names

```scala
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

```
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

dynamically constructing expressions

```scala
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

dynamically constructing expressions

```scala
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

dynamically constructing expressions

```scala
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

[left.a === right.a, left.b === right.b, ...]

```
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

left.a === right.a && left.b === right.b && ...

```
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

left.a === right.a && left.b === right.b && ...

```scala
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

```
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

dynamically constructing column lists

```
def natjoin(left: DataFrame, right: DataFrame): DataFrame = {
  val lcols = left.columns
  val rcols = right.columns
  val ccols = lcols.toSet intersect rcols.toSet

  if(ccols.isEmpty)
    left.limit(0).crossJoin(right.limit(0))
  else
    left
      .join(right, ccols.map {col => left(col) === right(col) }.reduce(_ && _))
      .select(lcols.collect { case c if ccols.contains(c) => left(c) } ++
              lcols.collect { case c if !ccols.contains(c) => left(c) } ++
              rcols.collect { case c if !ccols.contains(c) => right(c) } : _*)
}
```

dynamically constructing column lists

User-defined functions

```
{"a": 1, "b": "wilma", ..., "x": "club"}
{"a": 2, "b": "betty", ..., "x": "diamond"}
{"a": 3, "b": "fred", ..., "x": "heart"}
{"a": 4, "b": "barney", ..., "x": "spade"}
```

User-defined functions

```
{"a": 1, "b": "wilma", ..., "x": "club"}
{"a": 2, "b": "betty", ..., "x": "diamond"}
{"a": 3, "b": "fred", ..., "x": "heart"}
{"a": 4, "b": "barney", ..., "x": "spade"}
```

wilma	club
betty	diamond
fred	heart
barney	spade

```python
import json
from pyspark.sql.types import *
from pyspark.sql.functions import udf

def selectively_structure(fields):
  resultType = StructType([StructField(f, StringType(), nullable=True)
    for f in fields])
  def impl(js):
      try:
          d = json.loads(js)
          return [str(d.get(f)) for f in fields]
      except:
          return [None] * len(fields)
  return udf(impl, resultType)
```

```
import json
from pyspark.sql.types import *
from pyspark.sql.functions import udf

def selectively_structure(fields):
  resultType = StructType([StructField(f, StringType(), nullable=True)
    for f in fields])
  def impl(js):
      try:
          d = json.loads(js)
          return [str(d.get(f)) for f in fields]
      except:
          return [None] * len(fields)
  return udf(impl, resultType)
```

```python
import json
from pyspark.sql.types import *
from pyspark.sql.functions import udf

def selectively_structure(fields):
  resultType = StructType([StructField(f, StringType(), nullable=True)
    for f in fields])
  def impl(js):
      try:
          d = json.loads(js)
          return [str(d.get(f)) for f in fields]
      except:
          return [None] * len(fields)
  return udf(impl, resultType)
```

```
import json
from pyspark.sql.types import *
from pyspark.sql.functions import udf

def selectively_structure(fields):
  resultType = StructType([StructField(f, StringType(), nullable=True)
    for f in fields])
  def impl(js):
    try:
      d = json.loads(js)
      return [str(d.get(f)) for f in fields]
    except:
      return [None] * len(fields)
  return udf(impl, resultType)
```

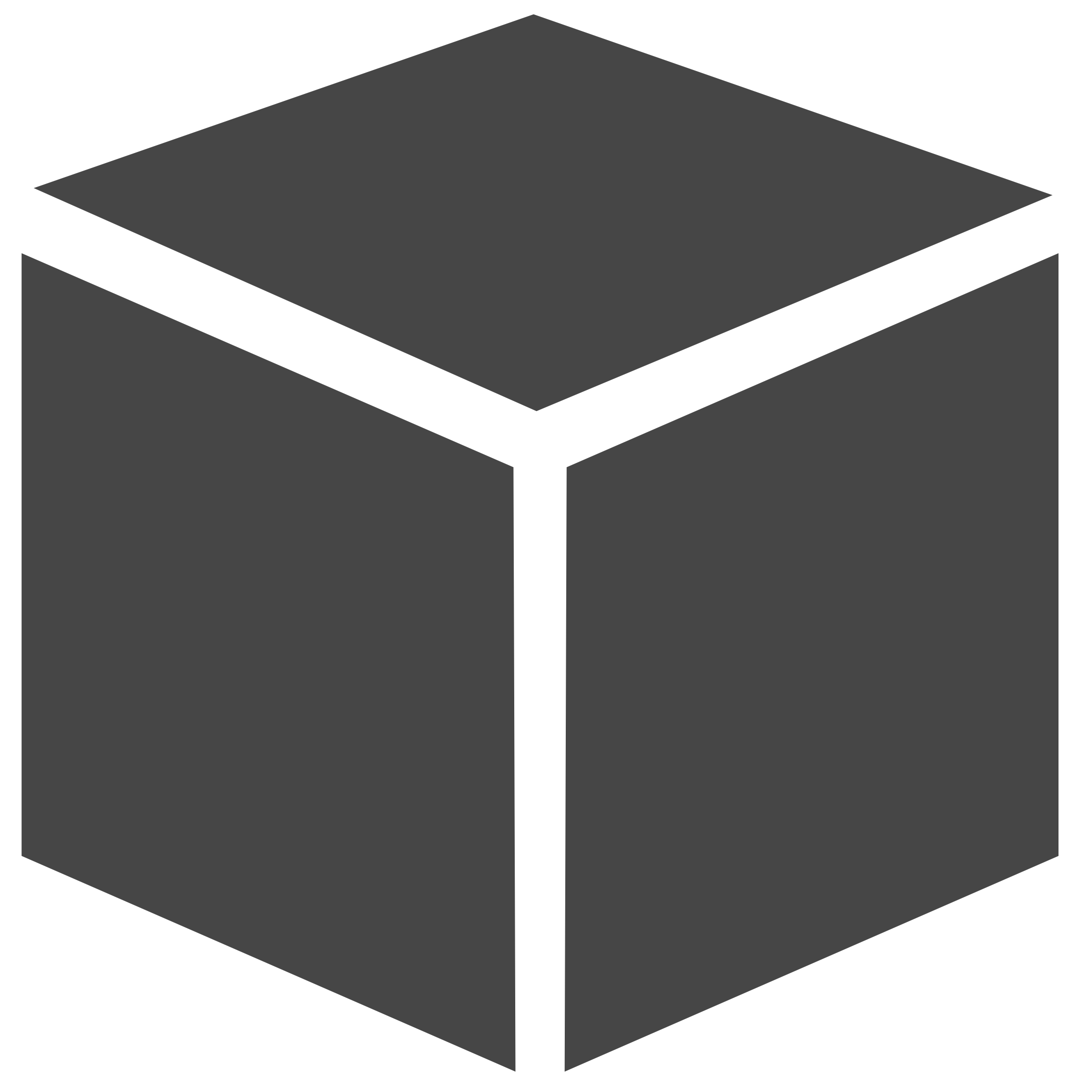
```
import json
from pyspark.sql.types import *
from pyspark.sql.functions import udf

def selectively_structure(fields):
  resultType = StructType([StructField(f, StringType(), nullable=True)
    for f in fields])
  def impl(js):
      try:
          d = json.loads(js)
          return [str(d.get(f)) for f in fields]
      except:
          return [None] * len(fields)
  return udf(impl, resultType)
```

```python
import json
from pyspark.sql.types import *
from pyspark.sql.functions import udf

def selectively_structure(fields):
  resultType = StructType([StructField(f, StringType(), nullable=True)
    for f in fields])
  def impl(js):
      try:
          d = json.loads(js)
          return [str(d.get(f)) for f in fields]
      except:
          return [None] * len(fields)
  return udf(impl, resultType)
```

```python
extract_bx = selectively_structure(["b", "x"])

structured_df = df.withColumn("result", extract_bx("json"))
```

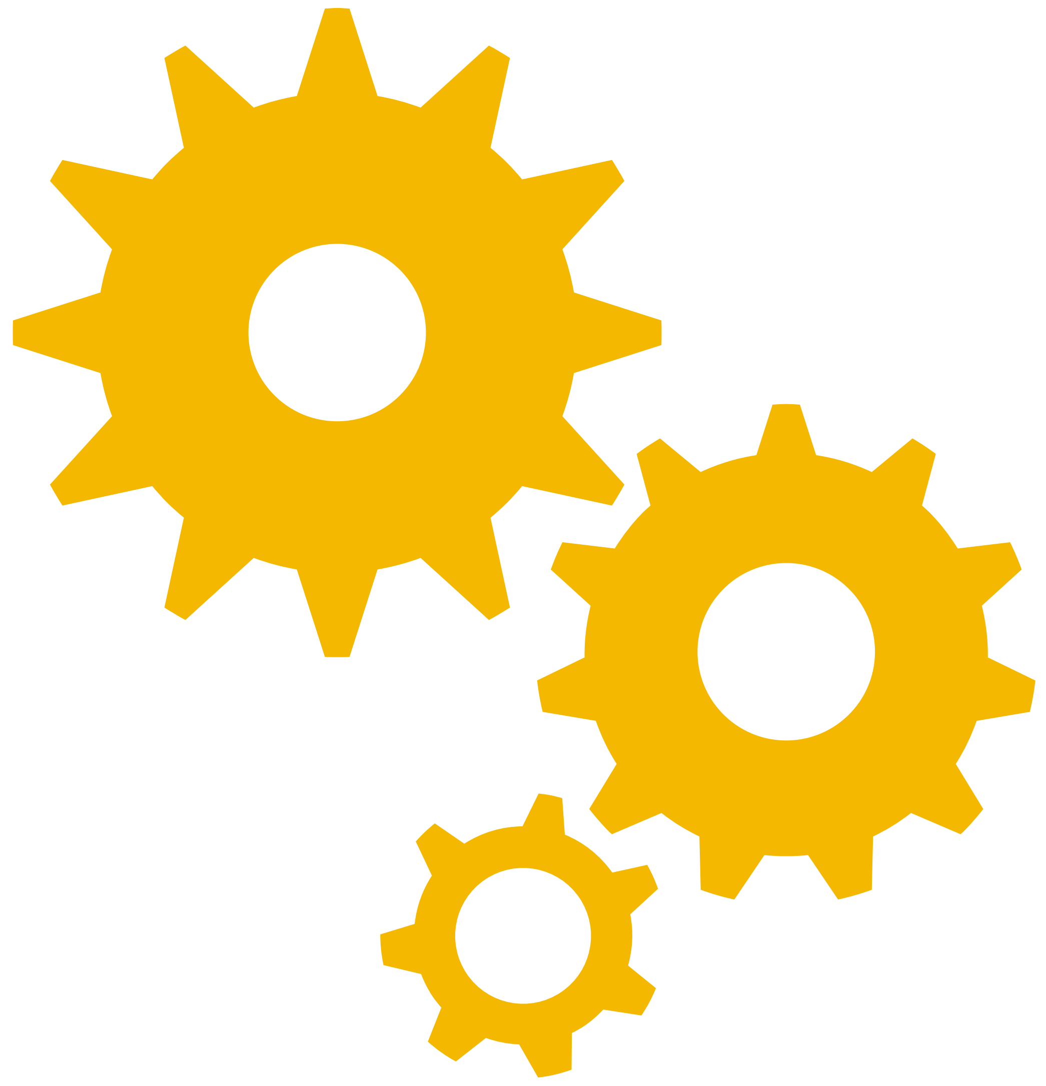
Spark’s ML pipelines

Spark's ML pipelines

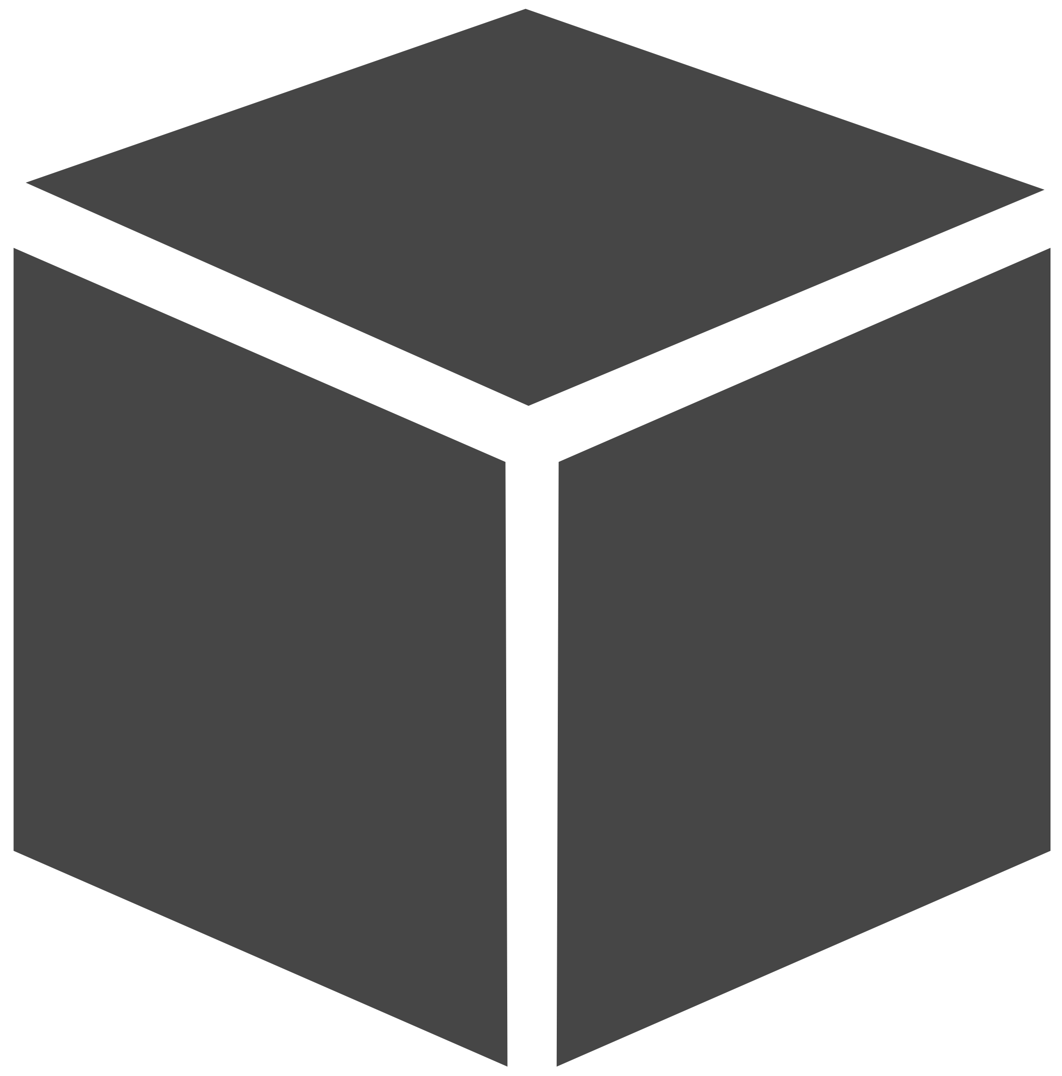


```
model.transform(df)
```

Spark's ML pipelines

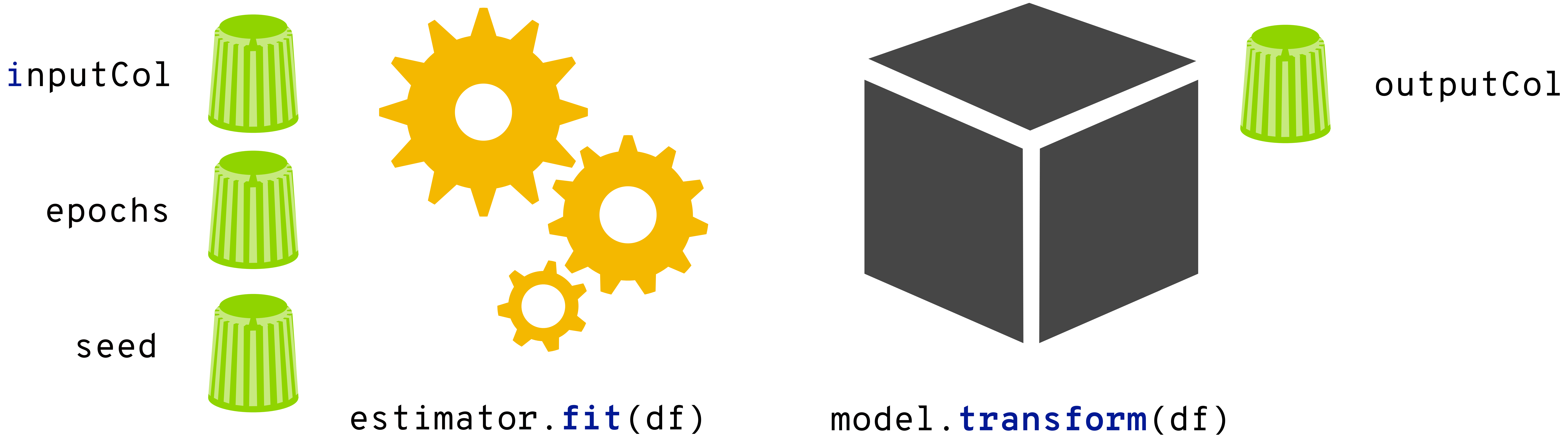


`estimator.fit(df)`



`model.transform(df)`

Spark's ML pipelines



Building Machine Learning Algorithms on Apache Spark: Scaling Out and Up

There are lots of reasons why you might want to implement your own machine learning algorithms on Spark: you might want to experiment with a new idea, try and reproduce results from a recent research paper, or simply to use an existing technique that isn't implemented in MLlib.

In this talk, we'll walk through the process of developing a new machine learning algorithm for Spark. We'll start with the basics, by considering how we'd design a scale-out parallel implementation of our unsupervised learning technique. The bulk of the talk will focus on the details you need to know to turn an algorithm design into an efficient parallel implementation on Spark.

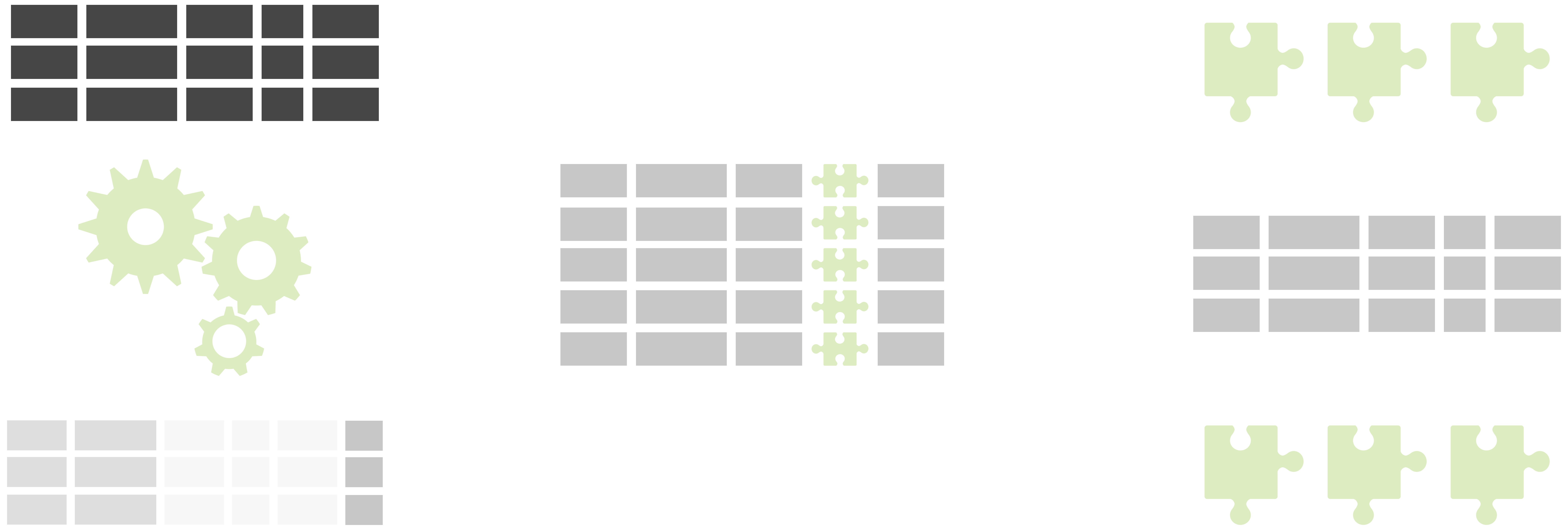
We'll start by reviewing a simple RDD-based implementation, show some improvements, point out some pitfalls to avoid, and iteratively extend our implementation to support contemporary Spark features like ML Pipelines and structured query processing. We'll conclude by briefly examining some useful techniques to complement scale-out performance by scaling our code up, taking advantage of specialized hardware to accelerate single-worker performance.

You'll leave this talk with everything you need to build a new machine learning technique that runs on Spark.

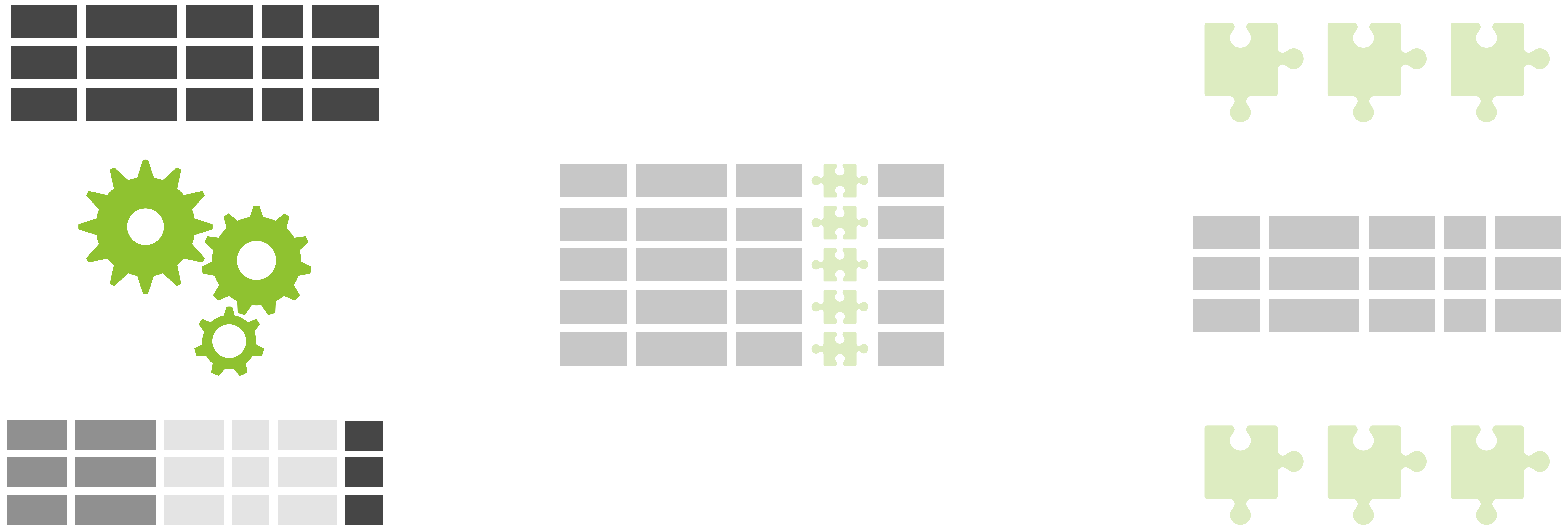
Session hashtag: #DS4SAIS

User-defined aggregates: the fundamentals

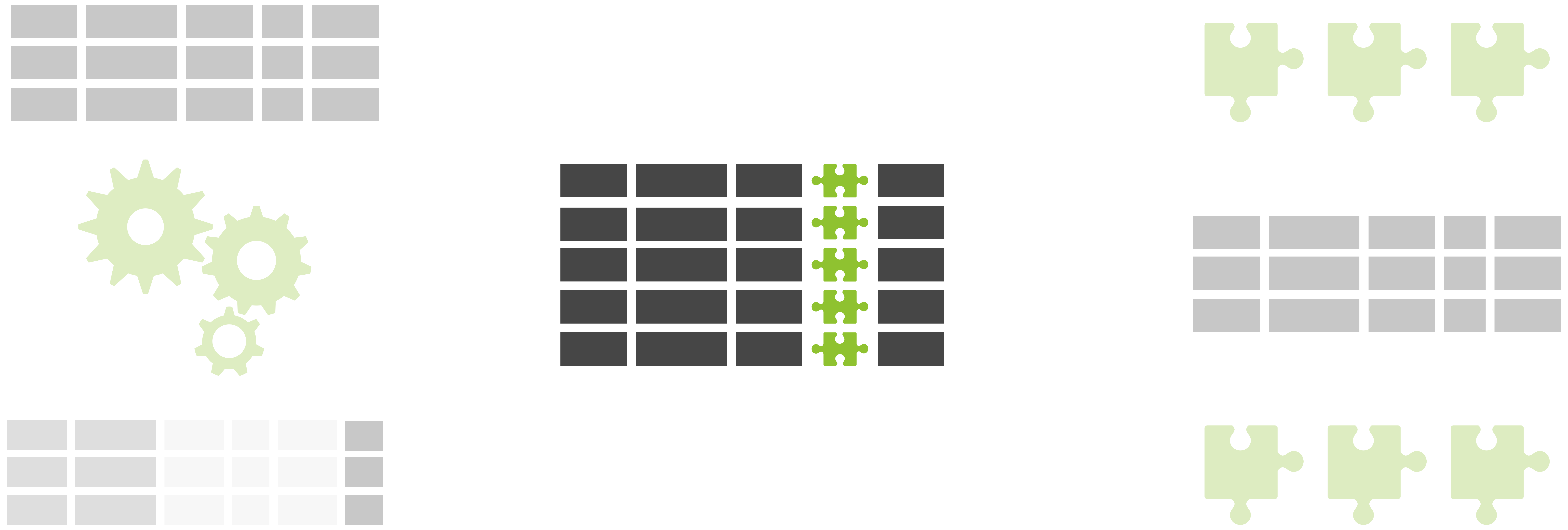
Three components



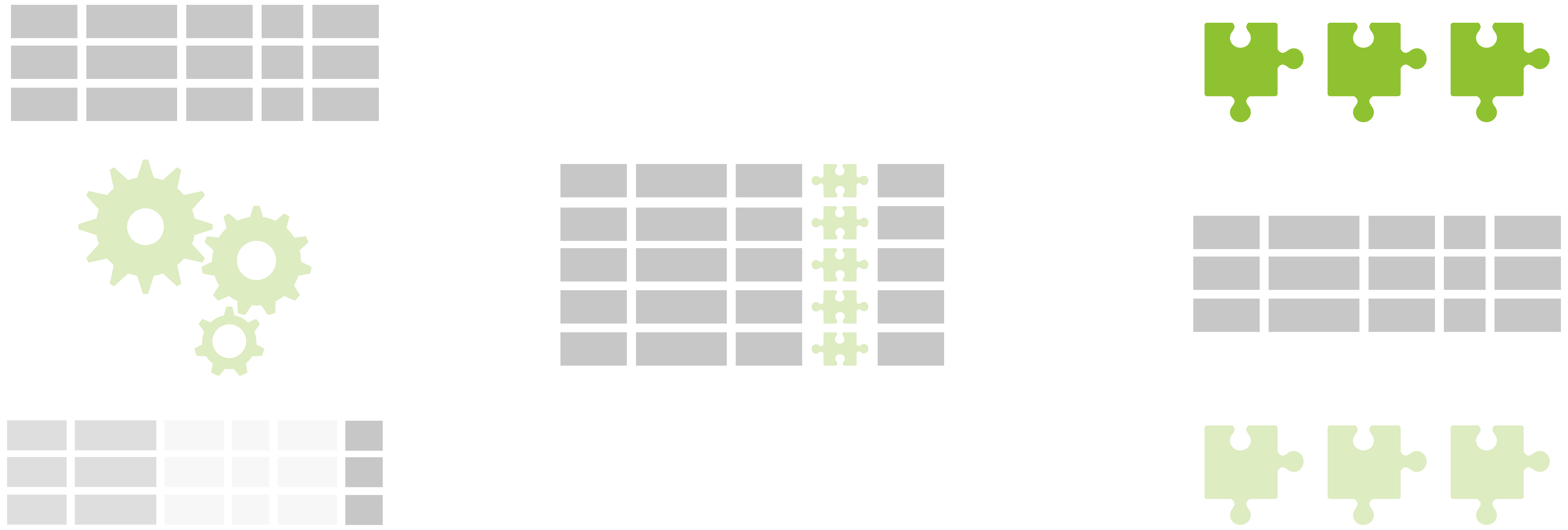
Three components



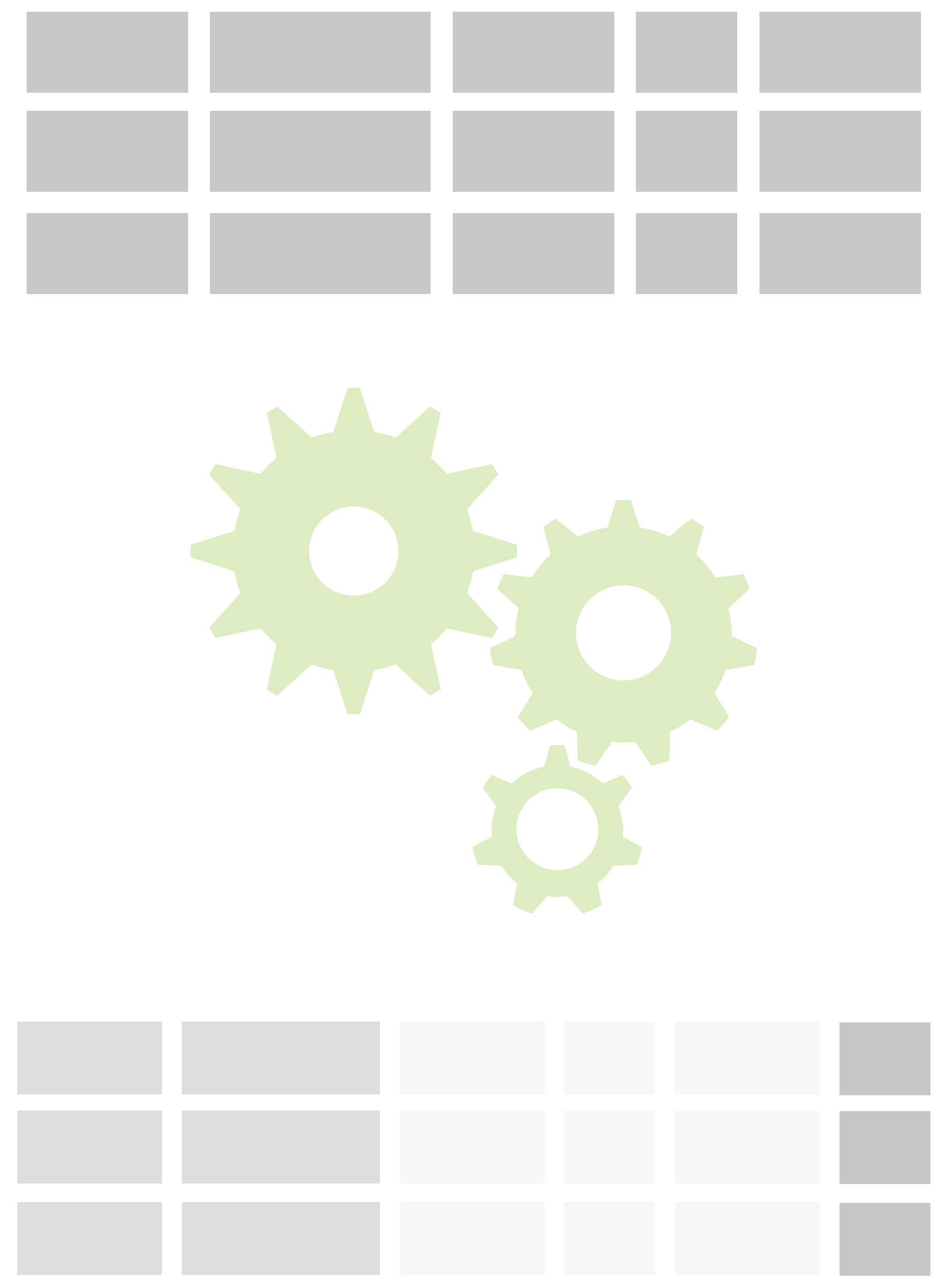
Three components

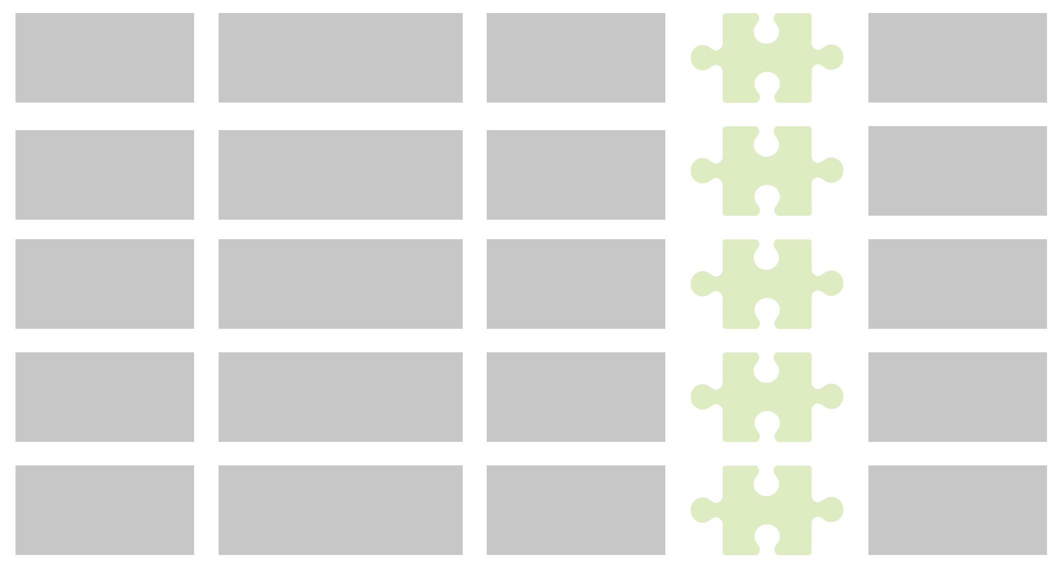


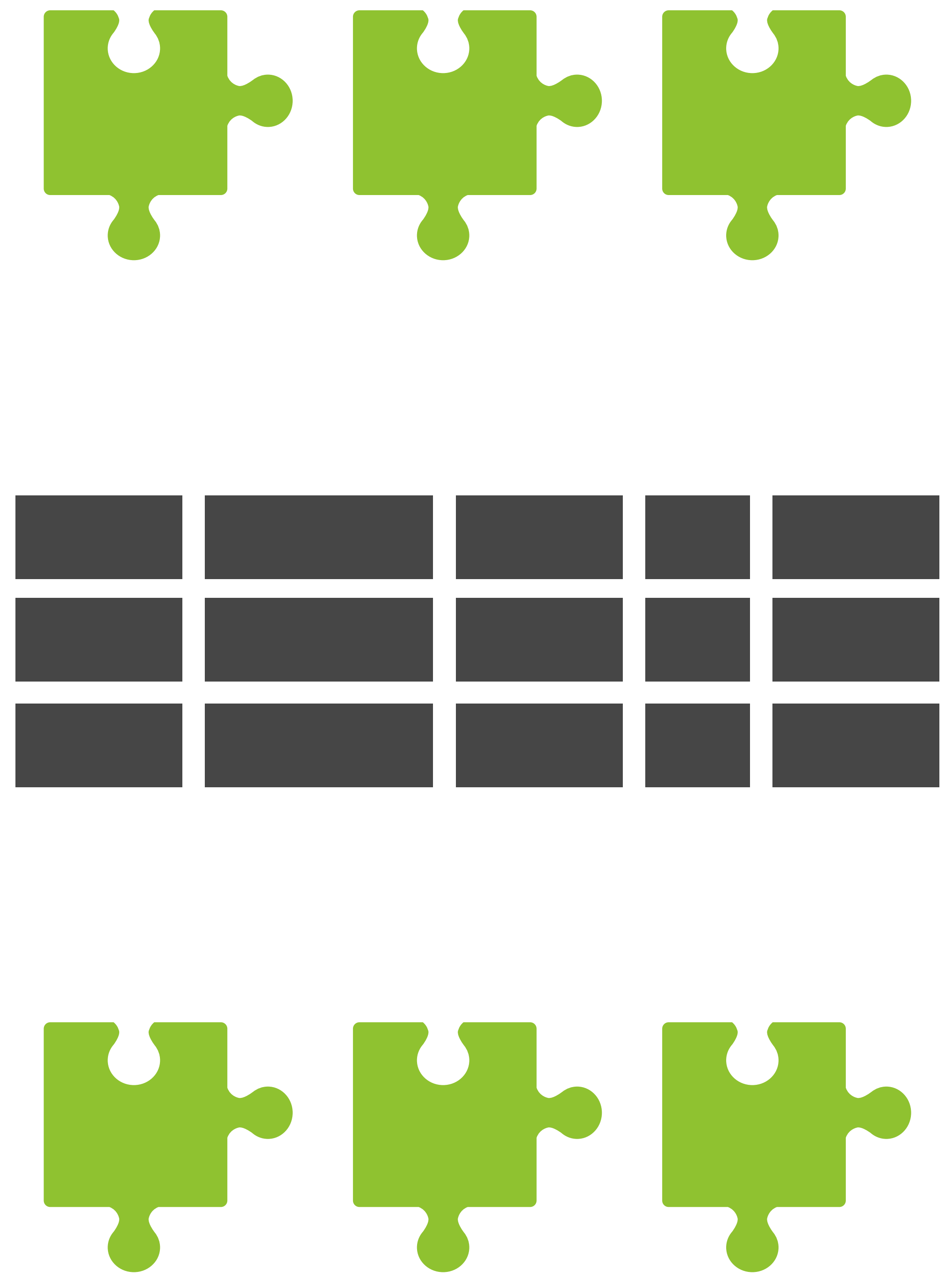
Three components



Three components







User-defined aggregates: the implementation

```scala
case class TDigestUDAF[N](deltaV: Double, maxDiscreteV: Int)
    (implicit num: Numeric[N], dataTpe: TDigestUDAFDataType[N])
  extends UserDefinedAggregateFunction {

    def deterministic: Boolean = false

    def inputSchema: StructType =
        StructType(StructField("x", dataTpe.tpe) :: Nil)

    def bufferSchema: StructType =
        StructType(StructField("tdigest", TDigestUDT) :: Nil)

    def dataType: DataType = TDigestUDT
```

```
case class TDigestUDAF[N](deltaV: Double, maxDiscreteV: Int)
    (implicit num: Numeric[N], dataTpe: TDigestUDAFDataType[N])
  extends UserDefinedAggregateFunction {

    def deterministic: Boolean = false

    def inputSchema: StructType =
        StructType(StructField("x", dataTpe.tpe) :: Nil)

    def bufferSchema: StructType =
        StructType(StructField("tdigest", TDigestUDT) :: Nil)

    def dataType: DataType = TDigestUDT
```

```scala
case class TDigestUDAF[N](deltaV: Double, maxDiscreteV: Int)
    (implicit num: Numeric[N], dataTpe: TDigestUDAFDataType[N])
  extends UserDefinedAggregateFunction {

    def deterministic: Boolean = false

    def inputSchema: StructType =
        StructType(StructField("x", dataTpe.tpe) :: Nil)

    def bufferSchema: StructType =
        StructType(StructField("tdigest", TDigestUDT) :: Nil)

    def dataType: DataType = TDigestUDT
```

```scala
case class TDigestUDAF[N](deltaV: Double, maxDiscreteV: Int)
    (implicit num: Numeric[N], dataTpe: TDigestUDAFDataType[N])
  extends UserDefinedAggregateFunction {

    def deterministic: Boolean = false

    def inputSchema: StructType =
        StructType(StructField("x", dataTpe.tpe) :: Nil)

    def bufferSchema: StructType =
        StructType(StructField("tdigest", TDigestUDT) :: Nil)

    def dataType: DataType = TDigestUDT
```

```
case class TDigestUDAF[N](deltaV: Double, maxDiscreteV: Int)
    (implicit num: Numeric[N], dataTpe: TDigestUDAFDataType[N])
  extends UserDefinedAggregateFunction {

    def deterministic: Boolean = false

    def inputSchema: StructType =
        StructType(StructField("x", dataTpe.tpe) :: Nil)

    def bufferSchema: StructType =
        StructType(StructField("tdigest", TDigestUDT) :: Nil)

    def dataType: DataType = TDigestUDT
```

```scala
case class TDigestUDAF[N](deltaV: Double, maxDiscreteV: Int)
    (implicit num: Numeric[N], dataTpe: TDigestUDAFDataType[N])
  extends UserDefinedAggregateFunction {

    def deterministic: Boolean = false

    def inputSchema: StructType =
        StructType(StructField("x", dataTpe.tpe) :: Nil)

    def bufferSchema: StructType =
        StructType(StructField("tdigest", TDigestUDT) :: Nil)

    def dataType: DataType = TDigestUDT
```

```scala
case class TDigestUDAF[N](deltaV: Double, maxDiscreteV: Int)
    (implicit num: Numeric[N], dataTpe: TDigestUDAFDataType[N])
  extends UserDefinedAggregateFunction {

    def deterministic: Boolean = false

    def inputSchema: StructType =
        StructType(StructField("x", dataTpe.tpe) :: Nil)

    def bufferSchema: StructType =
        StructType(StructField("tdigest", TDigestUDT) :: Nil)

    def dataType: DataType = TDigestUDT
```

```scala
case class TDigestUDAF[N](deltaV: Double, maxDiscreteV: Int)
    (implicit num: Numeric[N], dataTpe: TDigestUDAFDataType[N])
  extends UserDefinedAggregateFunction {

    def deterministic: Boolean = false

    def inputSchema: StructType =
        StructType(StructField("x", dataTpe.tpe) :: Nil)

    def bufferSchema: StructType =
        StructType(StructField("tdigest", TDigestUDT) :: Nil)

    def dataType: DataType = TDigestUDT
```

Four main functions: initialize

`initialize`

Four main functions: initialize

`initialize`

```
def initialize(buf: MutableAggregationBuffer): Unit = {
 buf(0) = TDigestSQL(TDigest.empty(deltaV, maxDiscreteV))
}

def evaluate(buf: Row): Any = buf.getAs[TDigestSQL](0)
```

```scala
def initialize(buf: MutableAggregationBuffer): Unit = {
 buf(0) = TDigestSQL(TDigest.empty(deltaV, maxDiscreteV))
}

def evaluate(buf: Row): Any = buf.getAs[TDigestSQL](0)
```

Four main functions: evaluate

`evaluate`

Four main functions: evaluate

`evaluate`

```
def initialize(buf: MutableAggregationBuffer): Unit = {
  buf(0) = TDigestSQL(TDigest.empty(deltaV, maxDiscreteV))
}

def evaluate(buf: Row): Any = buf.getAs[TDigestSQL](0)
def initialize(buf: MutableAggregationBuffer): Unit = {
  buf(0) = TDigestSQL(TDigest.empty(deltaV, maxDiscreteV))
}

def evaluate(buf: Row): Any = buf.getAs[TDigestSQL](0)
```

Four main functions: update

```
update
```

Four main functions: update

`update`

```
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

```scala
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

```scala
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

```
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

```
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

```
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

```
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

Four main functions: merge

`merge`

Four main functions: merge

1 + 2

`merge`

```scala
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

```scala
def update(buf: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
        buf(0) = TDigestSQL(buf.getAs[TDigestSQL](0).tdigest +
            num.toDouble(input.getAs[N](0)))
  }
}

def merge(buf1: MutableAggregationBuffer, buf2: Row): Unit = {
  buf1(0) = TDigestSQL(buf1.getAs[TDigestSQL](0).tdigest ++
        buf2.getAs[TDigestSQL](0).tdigest)
}
```

User-defined aggregates: User-defined types

User-defined types

```
package org.apache.spark.isarnproject.sketches.udt

@SQLUserDefinedType(udt = classOf[TDigestUDT])
case class TDigestSQL(tdigest: TDigest)

class TDigestUDT extends UserDefinedType[TDigestSQL] {
  def userClass: Class[TDigestSQL] = classOf[TDigestSQL]

  // ....
```

User-defined types

```
package org.apache.spark.isarnproject.sketches.udt

@SQLUserDefinedType(udt = classOf[TDigestUDT])
case class TDigestSQL(tdigest: TDigest)

class TDigestUDT extends UserDefinedType[TDigestSQL] {
  def userClass: Class[TDigestSQL] = classOf[TDigestSQL]

  // ....
```

User-defined types

```
package org.apache.spark.isarnproject.sketches.udt

@SQLUserDefinedType(udt = classOf[TDigestUDT])
case class TDigestSQL(tdigest: TDigest)

class TDigestUDT extends UserDefinedType[TDigestSQL] {
  def userClass: Class[TDigestSQL] = classOf[TDigestSQL]

  // ....
```

Implementing custom types

```
class TDigestUDT extends UserDefinedType[TDigestSQL] {
  def userClass: Class[TDigestSQL] = classOf[TDigestSQL]

  override def pyUDT: String =
    "isarnproject.sketches.udt.tdigest.TDigestUDT"

  override def typeName: String = "tdigest"

  def sqlType: DataType = StructType(
    StructField("delta", DoubleType, false) ::
   /* ... */
    StructField("clustM", ArrayType(DoubleType, false), false) ::
    Nil)
```

```scala
class TDigestUDT extends UserDefinedType[TDigestSQL] {
  def userClass: Class[TDigestSQL] = classOf[TDigestSQL]

  override def pyUDT: String =
    "isarnproject.sketches.udt.tdigest.TDigestUDT"

  override def typeName: String = "tdigest"

  def sqlType: DataType = StructType(
    StructField("delta", DoubleType, false) ::
   /* ... */
    StructField("clustM", ArrayType(DoubleType, false), false) ::
    Nil)
```

```
class TDigestUDT extends UserDefinedType[TDigestSQL] {
  def userClass: Class[TDigestSQL] = classOf[TDigestSQL]

  override def pyUDT: String =
    "isarnproject.sketches.udt.tdigest.TDigestUDT"

  override def typeName: String = "tdigest"

  def sqlType: DataType = StructType(
    StructField("delta", DoubleType, false) ::
   /* ... */
    StructField("clustM", ArrayType(DoubleType, false), false) ::
    Nil)
```

```scala
class TDigestUDT extends UserDefinedType[TDigestSQL] {
  def userClass: Class[TDigestSQL] = classOf[TDigestSQL]

  override def pyUDT: String =
    "isarnproject.sketches.udt.tdigest.TDigestUDT"

  override def typeName: String = "tdigest"

  def sqlType: DataType = StructType(
   StructField("delta", DoubleType, false) ::
   /* ... */
   StructField("clustM", ArrayType(DoubleType, false), false) ::
   Nil)
```

```scala
def serialize(tdsql: TDigestSQL): Any = serializeTD(tdsql.tdigest)

private[sketches] def serializeTD(td: TDigest): InternalRow = {
  val TDigest(delta, maxDiscrete, nclusters, clusters) = td
  val row = new GenericInternalRow(5)
  row.setDouble(0, delta)
  row.setInt(1, maxDiscrete)
  row.setInt(2, nclusters)
  val clustX = clusters.keys.toArray
  val clustM = clusters.values.toArray
  row.update(3, UnsafeArrayData.fromPrimitiveArray(clustX))
  row.update(4, UnsafeArrayData.fromPrimitiveArray(clustM))
  row
}
```

```scala
def serialize(tdsql: TDigestSQL): Any = serializeTD(tdsql.tdigest)

private[sketches] def serializeTD(td: TDigest): InternalRow = {
  val TDigest(delta, maxDiscrete, nclusters, clusters) = td
  val row = new GenericInternalRow(5)
  row.setDouble(0, delta)
  row.setInt(1, maxDiscrete)
  row.setInt(2, nclusters)
  val clustX = clusters.keys.toArray
  val clustM = clusters.values.toArray
  row.update(3, UnsafeArrayData.fromPrimitiveArray(clustX))
  row.update(4, UnsafeArrayData.fromPrimitiveArray(clustM))
  row
}
```

```scala
def serialize(tdsql: TDigestSQL): Any = serializeTD(tdsql.tdigest)

private[sketches] def serializeTD(td: TDigest): InternalRow = {
  val TDigest(delta, maxDiscrete, nclusters, clusters) = td
  val row = new GenericInternalRow(5)
  row.setDouble(0, delta)
  row.setInt(1, maxDiscrete)
  row.setInt(2, nclusters)
  val clustX = clusters.keys.toArray
  val clustM = clusters.values.toArray
  row.update(3, UnsafeArrayData.fromPrimitiveArray(clustX))
  row.update(4, UnsafeArrayData.fromPrimitiveArray(clustM))
  row
}
```

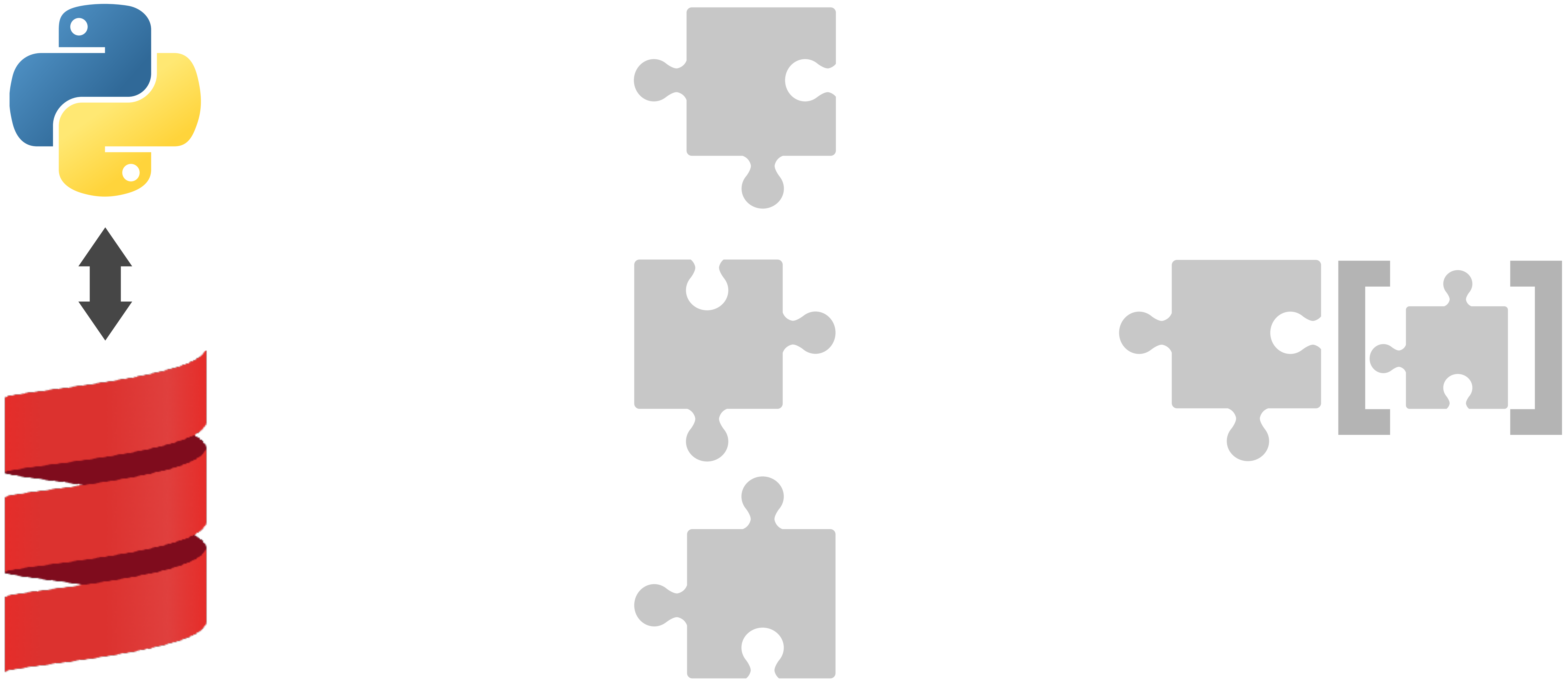
```scala
def serialize(tdsql: TDigestSQL): Any = serializeTD(tdsql.tdigest)

private[sketches] def serializeTD(td: TDigest): InternalRow = {
  val TDigest(delta, maxDiscrete, nclusters, clusters) = td
  val row = new GenericInternalRow(5)
  row.setDouble(0, delta)
  row.setInt(1, maxDiscrete)
  row.setInt(2, nclusters)
  val clustX = clusters.keys.toArray
  val clustM = clusters.values.toArray
  row.update(3, UnsafeArrayData.fromPrimitiveArray(clustX))
  row.update(4, UnsafeArrayData.fromPrimitiveArray(clustM))
  row
}
```

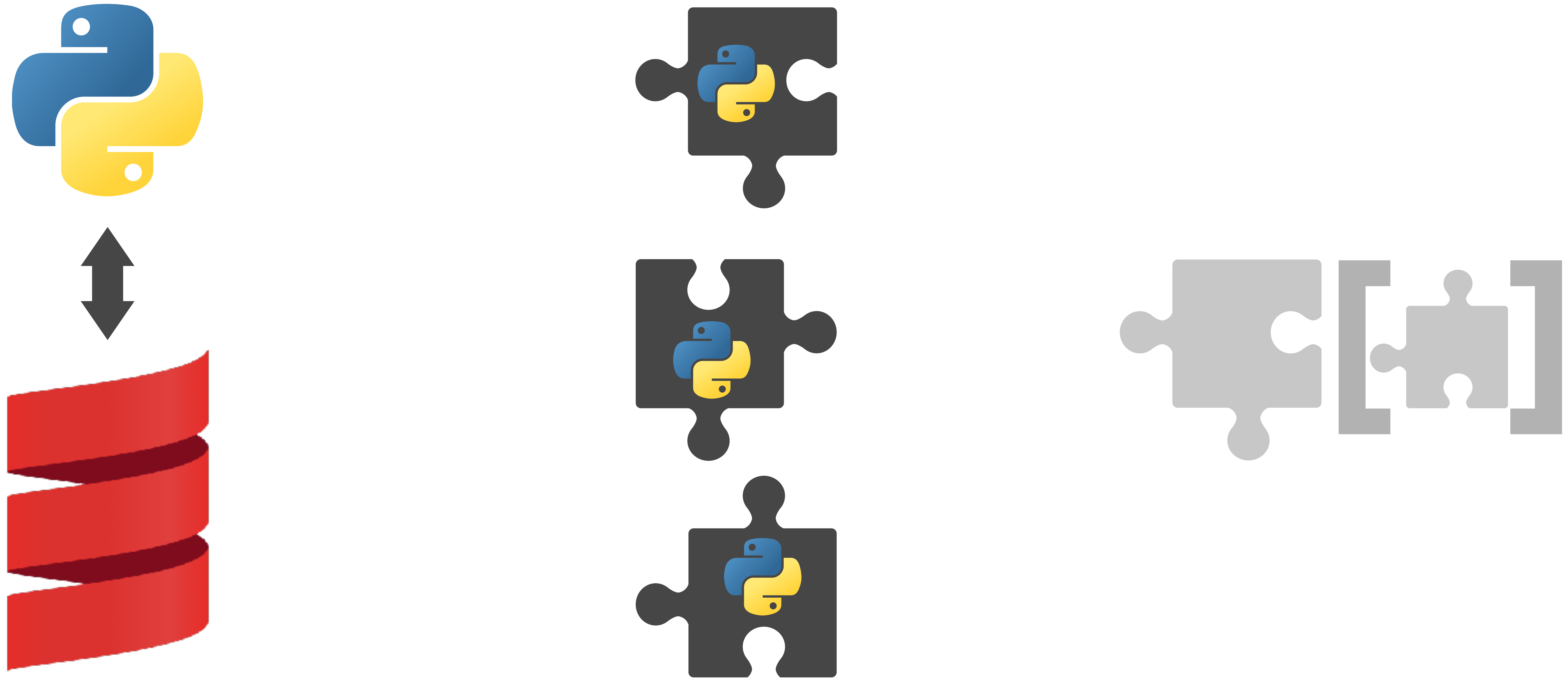
```scala
def deserialize(td: Any): TDigestSQL = TDigestSQL(deserializeTD(td))

private[sketches] def deserializeTD(datum: Any): TDigest =
  datum match { case row: InternalRow =>
   val delta = row.getDouble(0)
   val maxDiscrete = row.getInt(1)
   val nclusters = row.getInt(2)
   val clustX = row.getArray(3).toDoubleArray()
   val clustM = row.getArray(4).toDoubleArray()
   val clusters = clustX.zip(clustM)
     .foldLeft(TDigestMap.empty) { case (td, e) => td + e }
   TDigest(delta, maxDiscrete, nclusters, clusters)
}
```

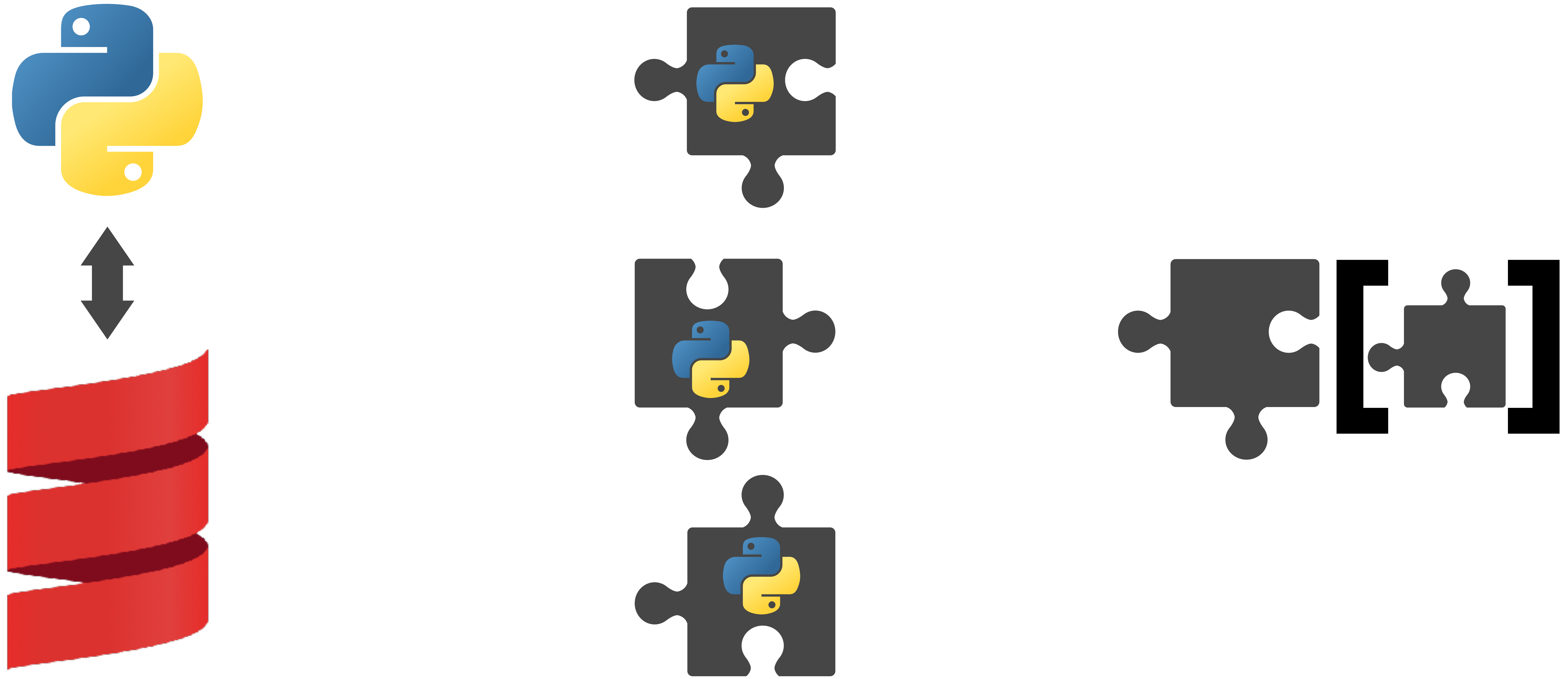
```scala
def deserialize(td: Any): TDigestSQL = TDigestSQL(deserializeTD(td))

private[sketches] def deserializeTD(datum: Any): TDigest =
  datum match { case row: InternalRow =>
    val delta = row.getDouble(0)
    val maxDiscrete = row.getInt(1)
    val nclusters = row.getInt(2)
    val clustX = row.getArray(3).toDoubleArray()
    val clustM = row.getArray(4).toDoubleArray()
    val clusters = clustX.zip(clustM)
      .foldLeft(TDigestMap.empty) { case (td, e) => td + e }
    TDigest(delta, maxDiscrete, nclusters, clusters)
}
```

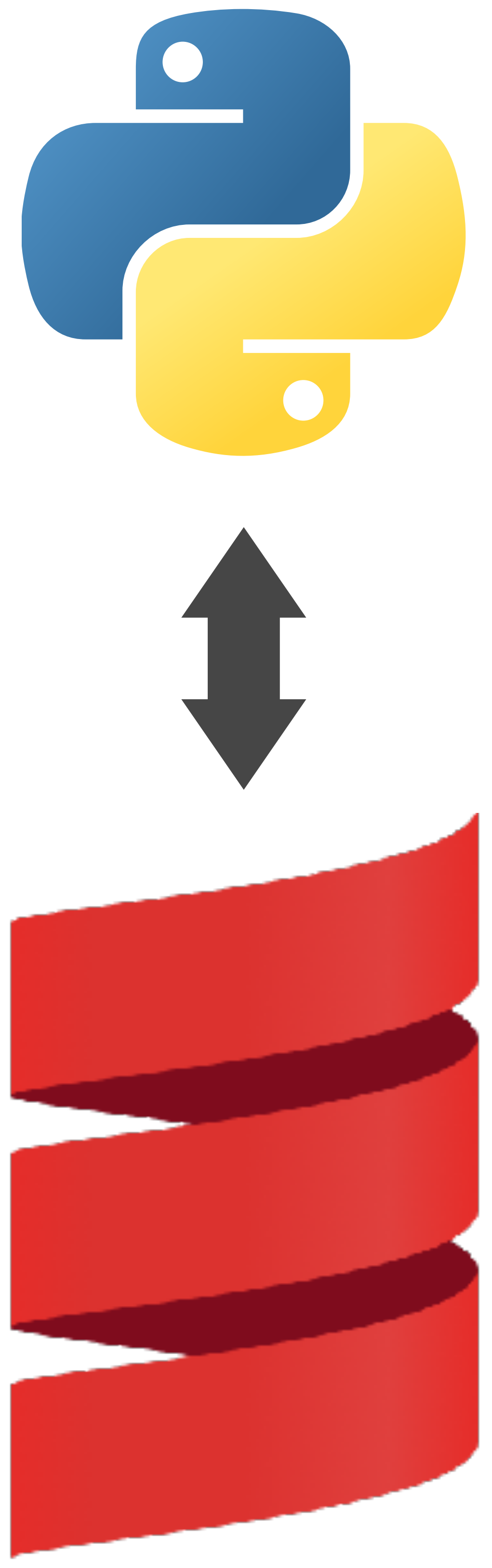
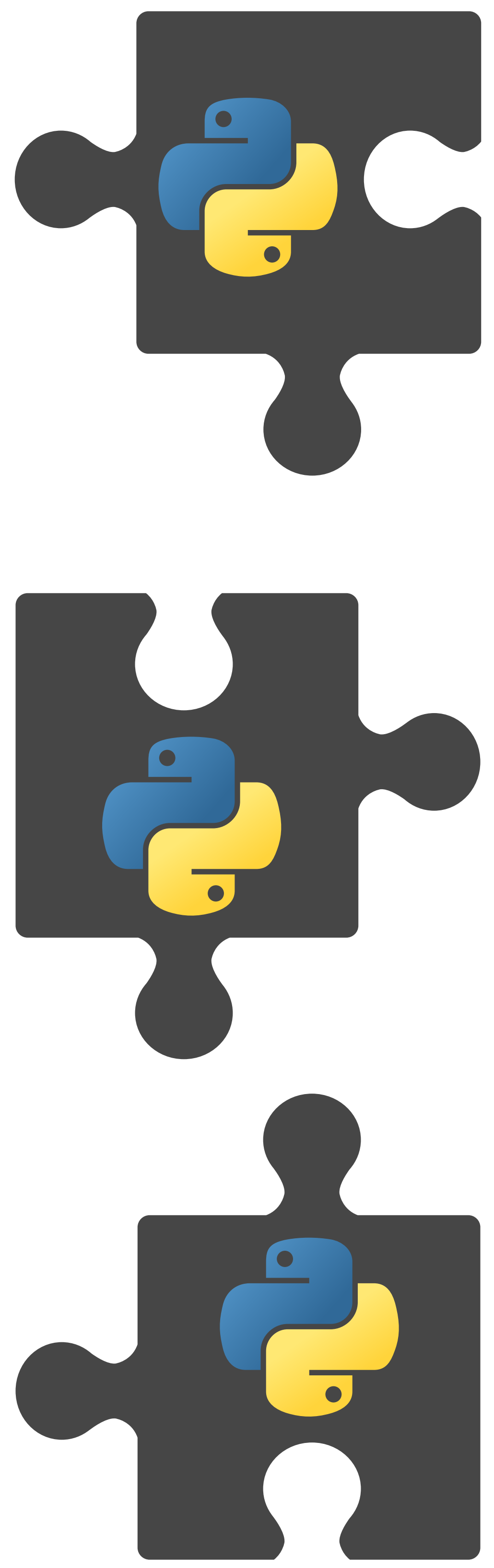
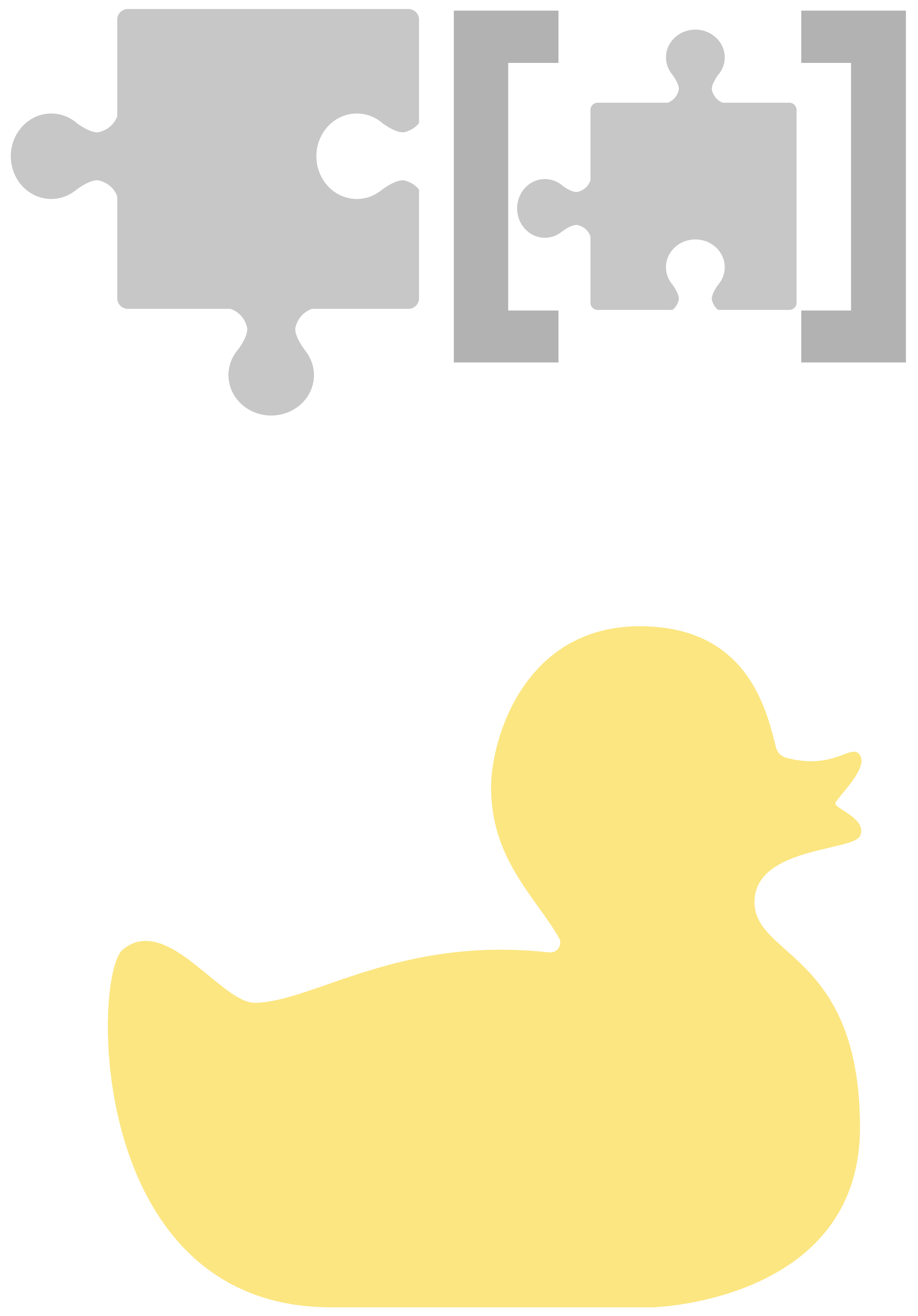
```scala
def deserialize(td: Any): TDigestSQL = TDigestSQL(deserializeTD(td))

private[sketches] def deserializeTD(datum: Any): TDigest =
  datum match { case row: InternalRow =>
   val delta = row.getDouble(0)
   val maxDiscrete = row.getInt(1)
   val nclusters = row.getInt(2)
   val clustX = row.getArray(3).toDoubleArray()
   val clustM = row.getArray(4).toDoubleArray()
   val clusters = clustX.zip(clustM)
     .foldLeft(TDigestMap.empty) { case (td, e) => td + e }
   TDigest(delta, maxDiscrete, nclusters, clusters)
}
```

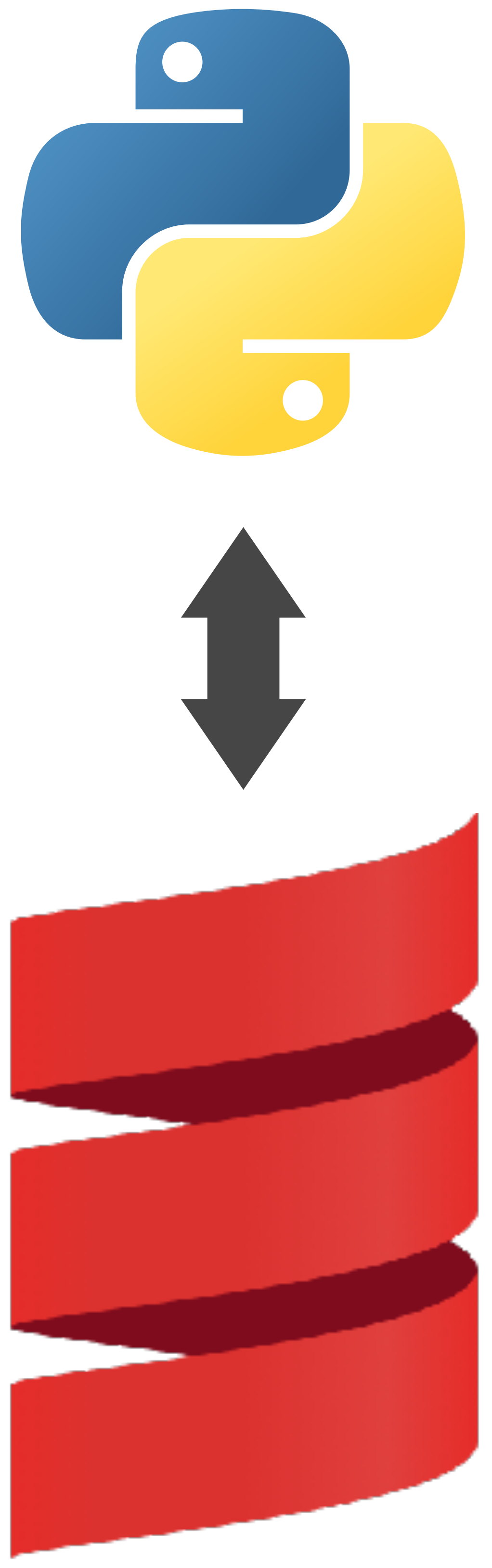
Extending PySpark with your Scala library



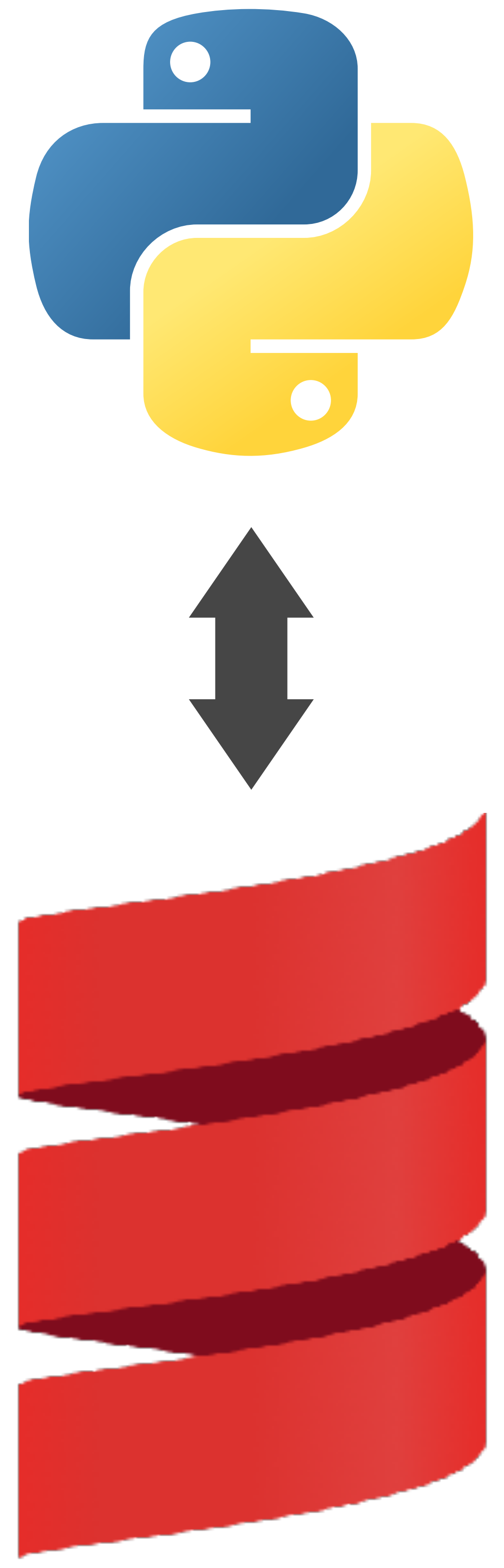




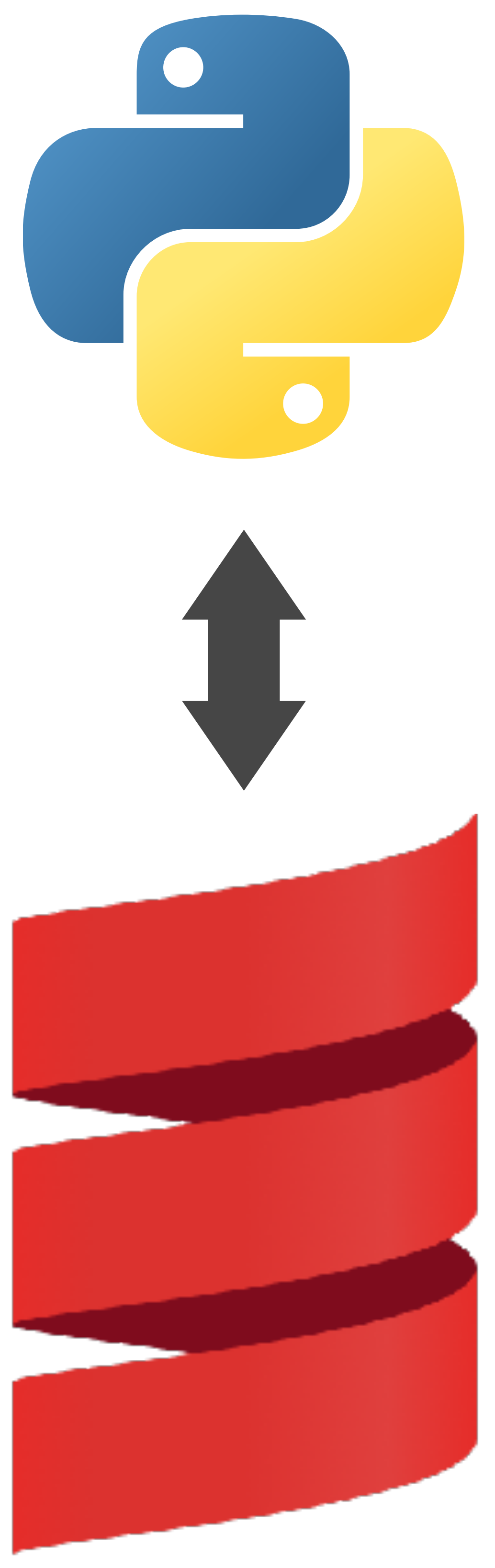






```
# class to access the active Spark context for Python
from pyspark.context import SparkContext

# gateway to the JVM from py4j
sparkJVM = SparkContext._active_spark_context._jvm

# use the gateway to access JVM objects and classes
thisThing = sparkJVM.com.path.to.this.thing
```



```
# class to access the active Spark context for Python
from pyspark.context import SparkContext

# gateway to the JVM from py4j
sparkJVM = SparkContext._active_spark_context._jvm

# use the gateway to access JVM objects and classes
thisThing = sparkJVM.com.path.to.this.thing
```



```
# class to access the active Spark context for Python
from pyspark.context import SparkContext

# gateway to the JVM from py4j
sparkJVM = SparkContext._active_spark_context._jvm

# use the gateway to access JVM objects and classes
thisThing = sparkJVM.com.path.to.this.thing
```

A Python-friendly wrapper

```
package org.isarnproject.sketches.udaf

object pythonBindings {
  def tdigestDoubleUDAF(delta: Double, maxDiscrete: Int) =
    TDigestUDAF[Double](delta, maxDiscrete)
}
```

```scala
package org.isarnproject.sketches.udaf

object pythonBindings {
  def tdigestDoubleUDAF(delta: Double, maxDiscrete: Int) =
    TDigestUDAF[Double](delta, maxDiscrete)
}
```

```scala
package org.isarnproject.sketches.udaf

object pythonBindings {
  def tdigestDoubleUDAF(delta: Double, maxDiscrete: Int) =
    TDigestUDAF[Double](delta, maxDiscrete)
}
```

```python
from pyspark.sql.column import Column, _to_java_column, _to_seq
from pyspark.context import SparkContext

# one of these for each type parameter Double, Int, Long, etc
def tdigestDoubleUDAF(col, delta=0.5, maxDiscrete=0):
  sc = SparkContext._active_spark_context
  pb = sc._jvm.org.isarnproject.sketches.udaf.pythonBindings
  tdapply = pb.tdigestDoubleUDAF(delta, maxDiscrete).apply
    return Column(tdapply(_to_seq(sc, [col], _to_java_column)))
```

```python
from pyspark.sql.column import Column, _to_java_column, _to_seq
from pyspark.context import SparkContext

# one of these for each type parameter Double, Int, Long, etc
def tdigestDoubleUDAF(col, delta=0.5, maxDiscrete=0):
  sc = SparkContext._active_spark_context
  pb = sc._jvm.org.isarnproject.sketches.udaf.pythonBindings
  tdapply = pb.tdigestDoubleUDAF(delta, maxDiscrete).apply
    return Column(tdapply(_to_seq(sc, [col], _to_java_column)))
```

```
from pyspark.sql.column import Column, _to_java_column, _to_seq
from pyspark.context import SparkContext

# one of these for each type parameter Double, Int, Long, etc
def tdigestDoubleUDAF(col, delta=0.5, maxDiscrete=0):
  sc = SparkContext._active_spark_context
  pb = sc._jvm.org.isarnproject.sketches.udaf.pythonBindings
  tdapply = pb.tdigestDoubleUDAF(delta, maxDiscrete).apply
    return Column(tdapply(_to_seq(sc, [col], _to_java_column)))
```

```
class TDigestUDT(UserDefinedType):
  @classmethod
  def sqlType(cls):
    return StructType([
      StructField("delta", DoubleType(), False),
      StructField("maxDiscrete", IntegerType(), False),
      StructField("nclusters", IntegerType(), False),
      StructField("clustX", ArrayType(DoubleType(), False), False),
      StructField("clustM", ArrayType(DoubleType(), False), False)])

  # ...
```

```
class TDigestUDT(UserDefinedType):
  @classmethod
  def sqlType(cls):
    return StructType([
      StructField("delta", DoubleType(), False),
      StructField("maxDiscrete", IntegerType(), False),
      StructField("nclusters", IntegerType(), False),
      StructField("clustX", ArrayType(DoubleType(), False), False),
      StructField("clustM", ArrayType(DoubleType(), False), False)])

  # ...
```

```
class TDigestUDT(UserDefinedType):
  # ...
  @classmethod
  def module(cls):
    return "isarnproject.sketches.udt.tdigest"

  @classmethod
  def scalaUDT(cls):
    return "org.apache.spark.isarnproject.sketches.udt.TDigestUDT"

  def simpleString(self):
    return "tdigest"
```

```python
class TDigestUDT(UserDefinedType):
  # ...
  @classmethod
  def module(cls):
    return "isarnproject.sketches.udt.tdigest"

  @classmethod
  def scalaUDT(cls):
    return "org.apache.spark.isarnproject.sketches.udt.TDigestUDT"

  def simpleString(self):
    return "tdigest"
```

```python
class TDigestUDT(UserDefinedType):
  # ...
  @classmethod
  def module(cls):
    return "isarnproject.sketches.udt.tdigest"

  @classmethod
  def scalaUDT(cls):
    return "org.apache.spark.isarnproject.sketches.udt.TDigestUDT"

  def simpleString(self):
    return "tdigest"
```

```
class TDigestUDT(UserDefinedType):
  # ...
  def serialize(self, obj):
    return (obj.delta, obj.maxDiscrete, obj.nclusters, \
      [float(v) for v in obj.clustX], \
      [float(v) for v in obj.clustM])

 def deserialize(self, datum):
    return TDigest(datum[0], datum[1], datum[2], datum[3], datum[4])
```

```
class TDigestUDT(UserDefinedType):
  # ...
  def serialize(self, obj):
    return (obj.delta, obj.maxDiscrete, obj.nclusters, \
      [float(v) for v in obj.clustX], \
      [float(v) for v in obj.clustM])

 def deserialize(self, datum):
    return TDigest(datum[0], datum[1], datum[2], datum[3], datum[4])
```

```
class TDigestUDT extends UserDefinedType[TDigestSQL] {
 // ...
 override def pyUDT: String =
   “isarnproject.sketches.udt.tdigest.TDigestUDT"
}
```

Python code in JAR files

```
mappings in (Compile, packageBin) ++= Seq(
  (baseDirectory.value / "python" / "isarnproject" / "__init__.pyc") ->
    "isarnproject/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "__init__.pyc") ->
    "isarnproject/sketches/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udaf" / "__init__.pyc") ->
    "isarnproject/sketches/udaf/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udaf" / "tdigest.pyc") ->
    "isarnproject/sketches/udaf/tdigest.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udt" / "__init__.pyc") ->
    "isarnproject/sketches/udt/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udt" / "tdigest.pyc") ->
    "isarnproject/sketches/udt/tdigest.pyc"
)
```

```
mappings in (Compile, packageBin) ++= Seq(
  (baseDirectory.value / "python" / "isarnproject" / "__init__.pyc") ->
    "isarnproject/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "__init__.pyc") ->
    "isarnproject/sketches/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udaf" / "__init__.pyc") ->
    "isarnproject/sketches/udaf/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udaf" / "tdigest.pyc") ->
    "isarnproject/sketches/udaf/tdigest.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udt" / "__init__.pyc") ->
    "isarnproject/sketches/udt/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udt" / "tdigest.pyc") ->
    "isarnproject/sketches/udt/tdigest.pyc"
)
```

```
mappings in (Compile, packageBin) ++= Seq(
  (baseDirectory.value / "python" / "isarnproject" / "__init__.pyc") ->
    "isarnproject/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "__init__.pyc") ->
    "isarnproject/sketches/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udaf" / "__init__.pyc") ->
    "isarnproject/sketches/udaf/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udaf" / "tdigest.pyc") ->
    "isarnproject/sketches/udaf/tdigest.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udt" / "__init__.pyc") ->
    "isarnproject/sketches/udt/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udt" / "tdigest.pyc") ->
    "isarnproject/sketches/udt/tdigest.pyc"
)
```

```
mappings in (Compile, packageBin) ++= Seq(
  (baseDirectory.value / "python" / "isarnproject" / "__init__.pyc") ->
    "isarnproject/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "__init__.pyc") ->
    "isarnproject/sketches/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udaf" / "__init__.pyc") ->
    "isarnproject/sketches/udaf/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udaf" / "tdigest.pyc") ->
    "isarnproject/sketches/udaf/tdigest.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udt" / "__init__.pyc") ->
    "isarnproject/sketches/udt/__init__.pyc",
  (baseDirectory.value / "python" / "isarnproject" / "sketches" / "udt" / "tdigest.pyc") ->
    "isarnproject/sketches/udt/tdigest.pyc"
)
```

Cross-building for Python

```
lazy val compilePython = taskKey[Unit]("Compile python files")

compilePython := {
  val s: TaskStreams = streams.value
  s.log.info("compiling python...")
  val stat = (Seq(pythonCMD, "-m", "compileall", "python/") !)
  if (stat != 0) {
  throw new IllegalStateException("python compile failed")
  }
}

(packageBin in Compile) <<=
  (packageBin in Compile).dependsOn(compilePython)
```

```
lazy val compilePython = taskKey[Unit]("Compile python files")

compilePython := {
  val s: TaskStreams = streams.value
  s.log.info("compiling python...")
  val stat = (Seq(pythonCMD, "-m", "compileall", "python/") !)
  if (stat != 0) {
 throw new IllegalStateException("python compile failed")
  }
}

(packageBin in Compile) <<=
  (packageBin in Compile).dependsOn(compilePython)
```

```
lazy val compilePython = taskKey[Unit]("Compile python files")

compilePython := {
  val s: TaskStreams = streams.value
  s.log.info("compiling python...")
  val stat = (Seq(pythonCMD, "-m", "compileall", "python/") !)
  if (stat != 0) {
 throw new IllegalStateException("python compile failed")
  }
}

(packageBin in Compile) <<=
  (packageBin in Compile).dependsOn(compilePython)
```

Using versioned JAR files

```
$ pyspark --packages \
    'org.isarnproject:isarn-sketches-spark_2.11:0.3.0-sp2.2-py2.7'
```

Using versioned JAR files

```
$ pyspark --packages \
    'org.isarnproject:isarn-sketches-spark_2.11:0.3.0-sp2.2-py2.7'
```

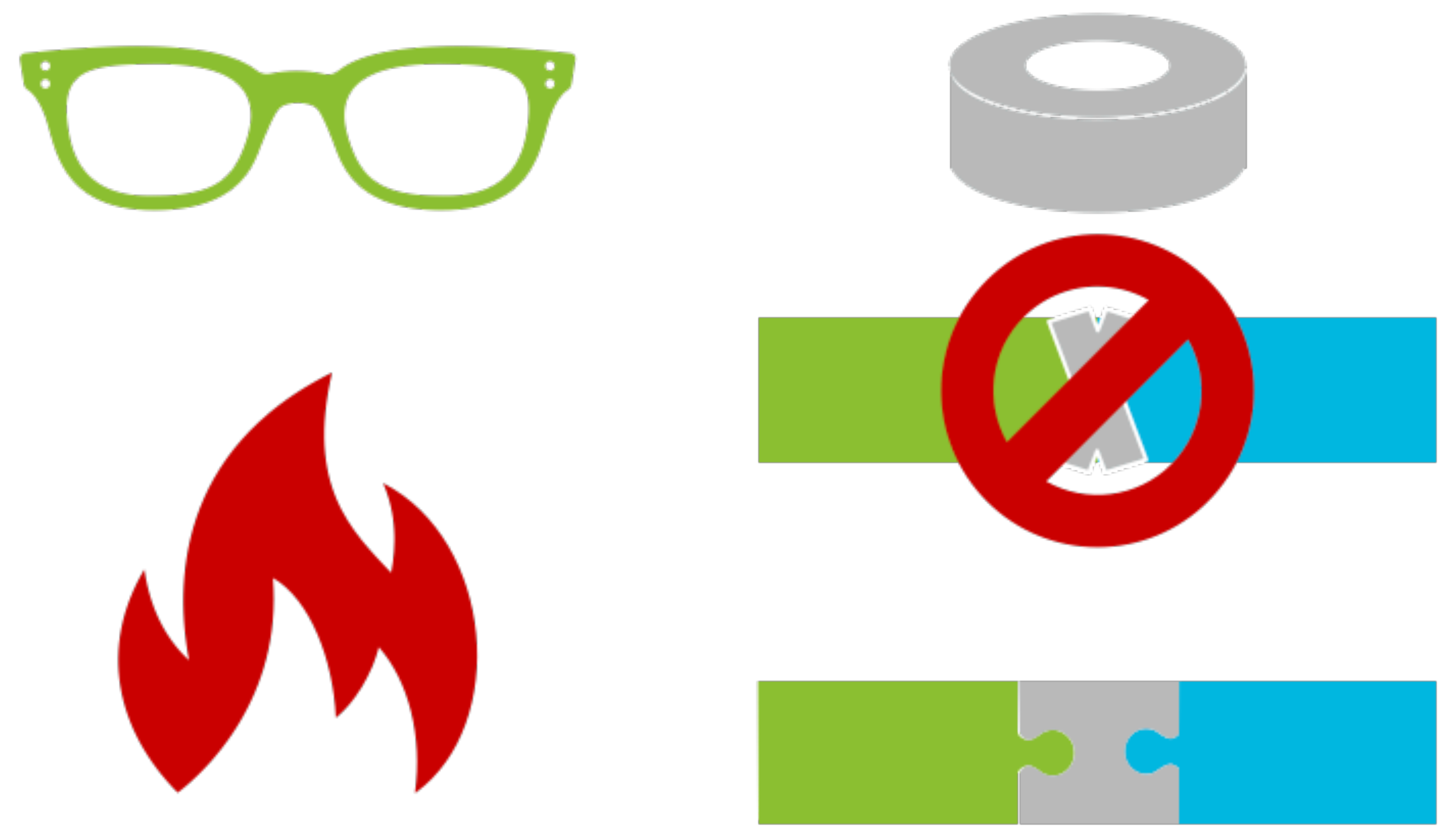
Using versioned JAR files

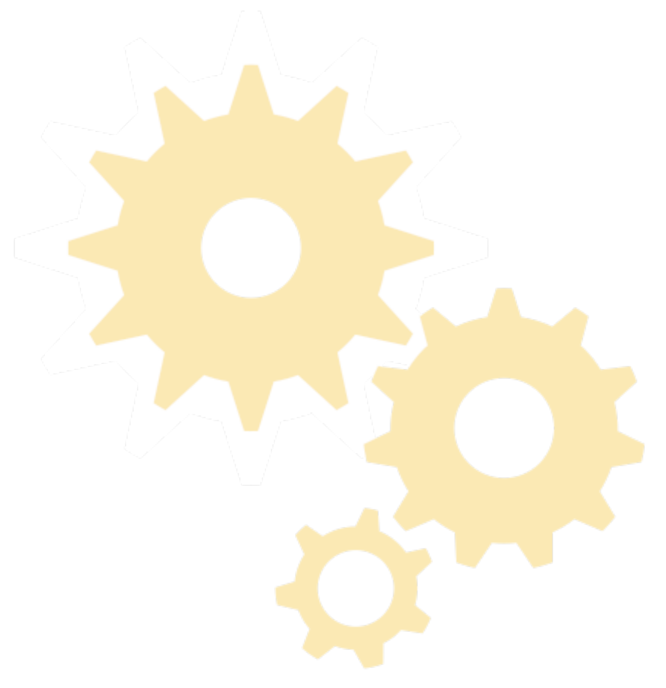
```
$ pyspark --packages \
    'org.isarnproject:isarn-sketches-spark_2.11:0.3.0-sp2.2-py2.7'
```

Show your work: publishing results

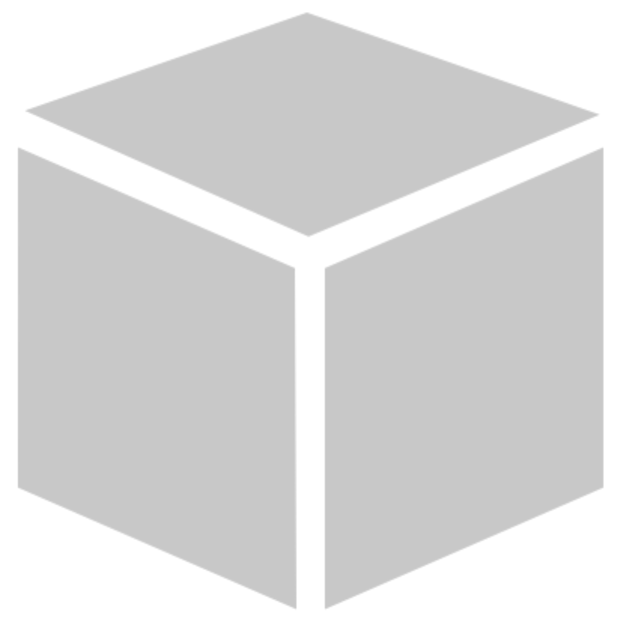
Maven Central		Bintray
not really	**easy to set up for library developers**	trivial
trivial	**easy to set up for library users**	mostly
yes, via sbt	**easy to publish**	yes, via sbt + plugins
yes	**easy to resolve artifacts**	mostly

Conclusions and takeaways

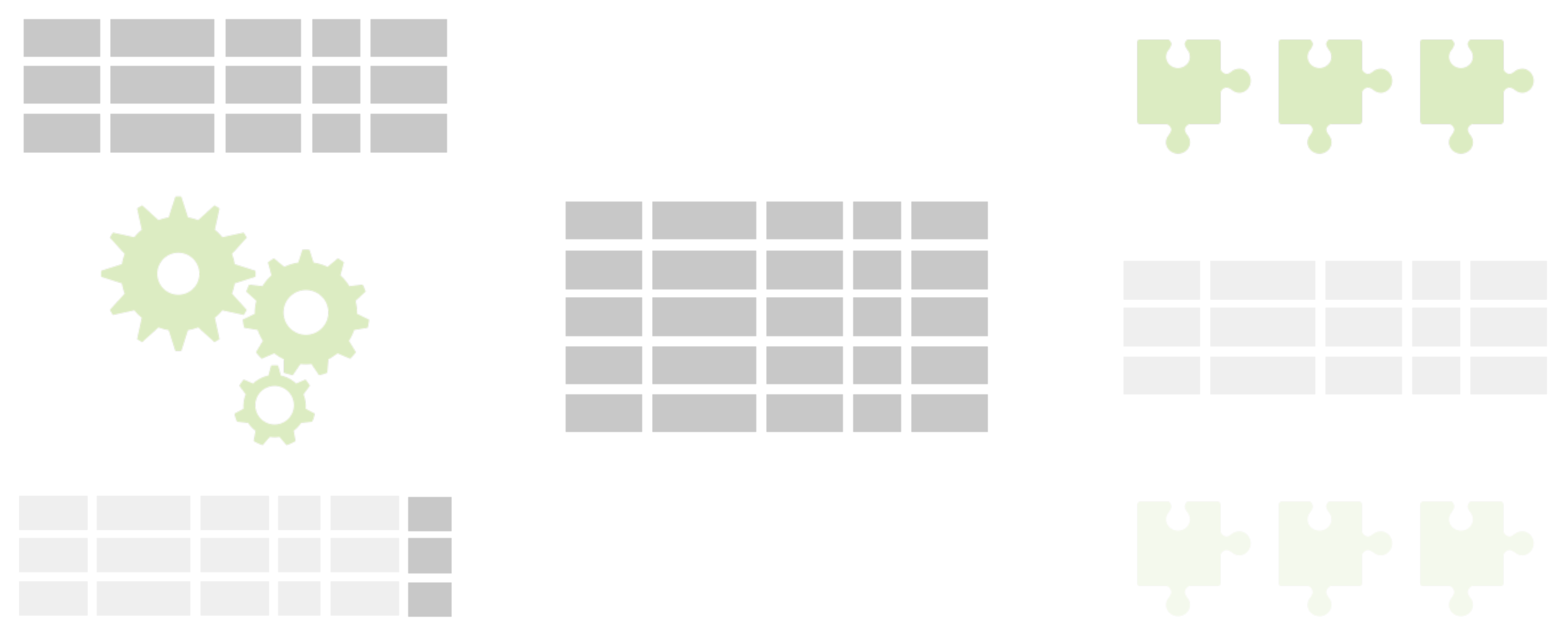


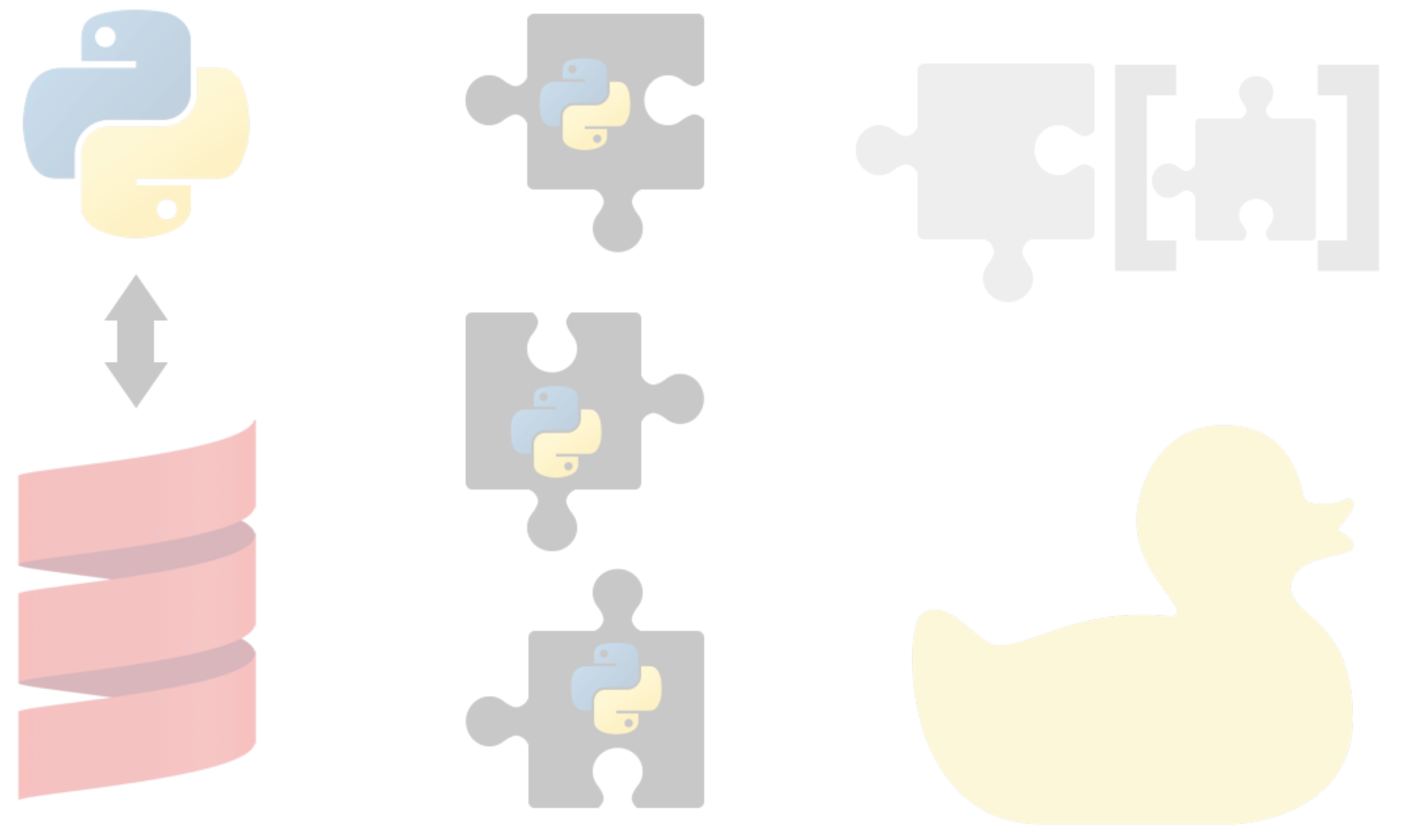


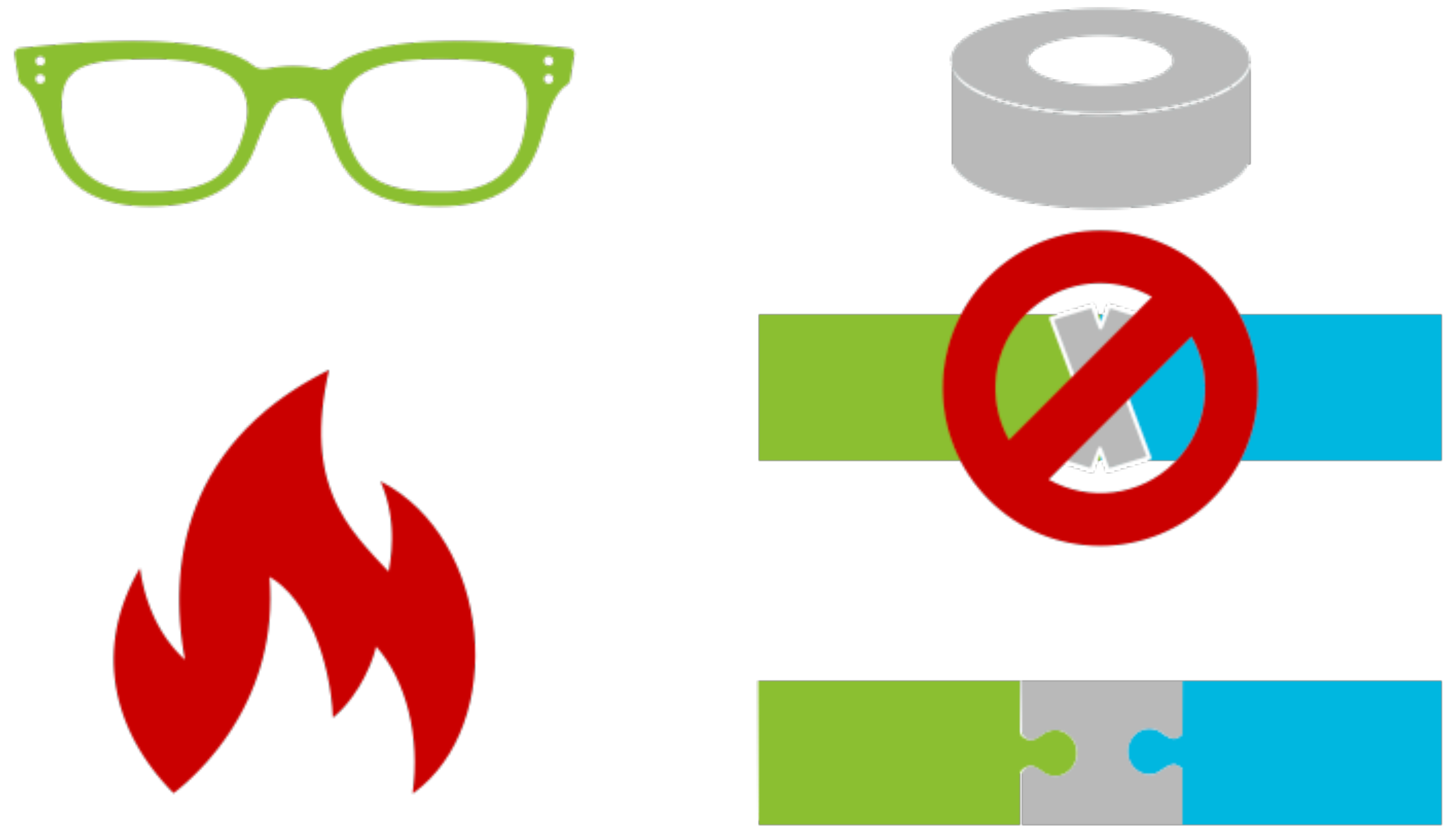
estimator.fit(df)

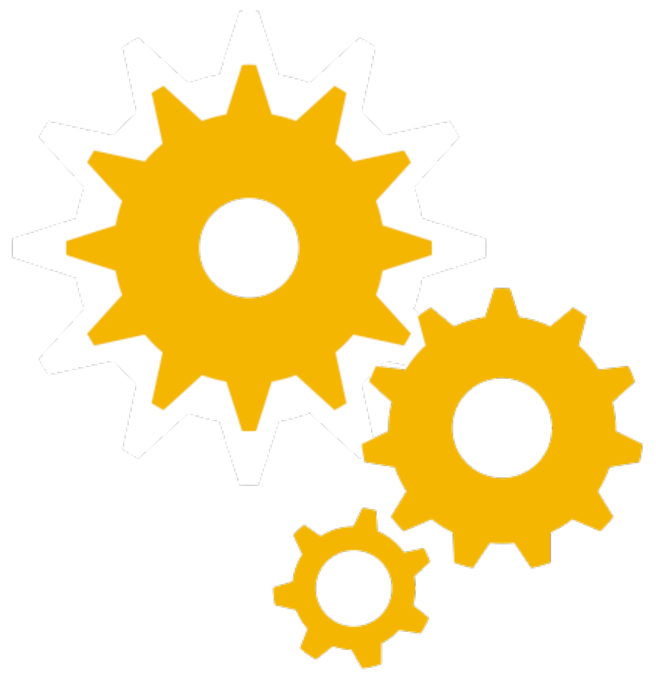


model.transform(df)

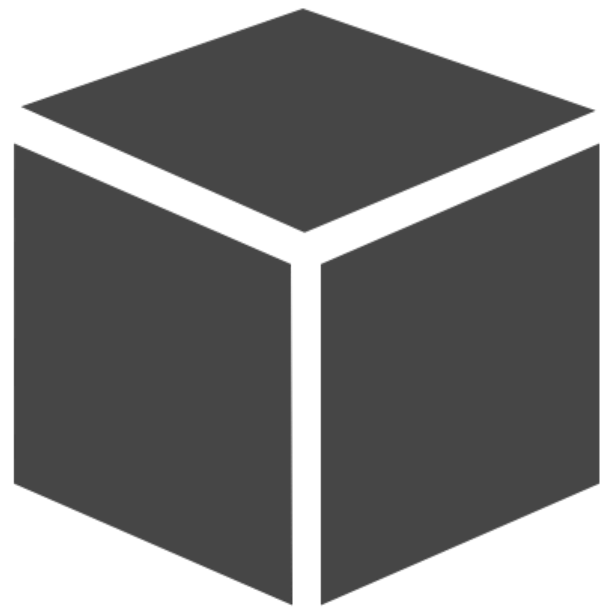




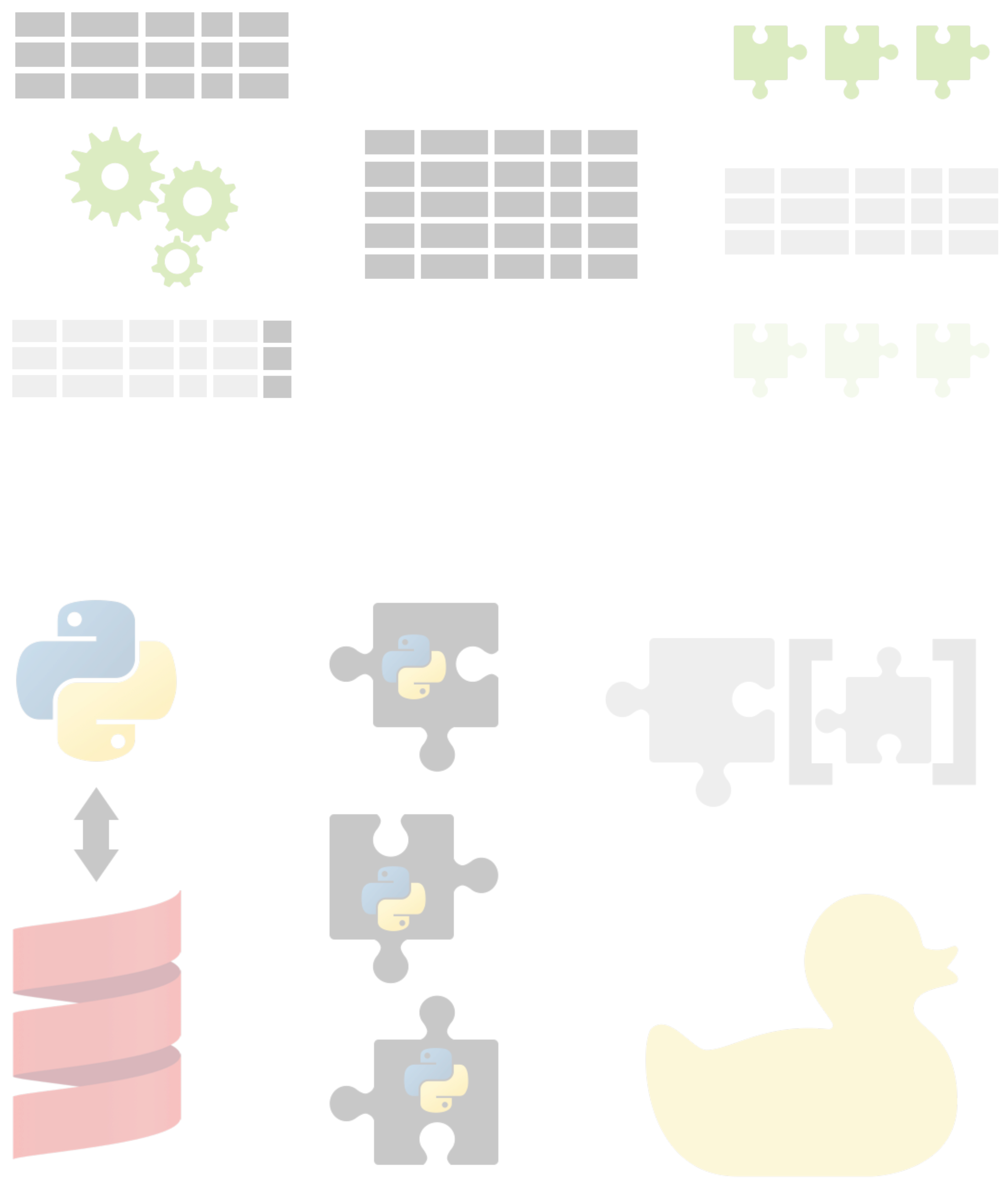


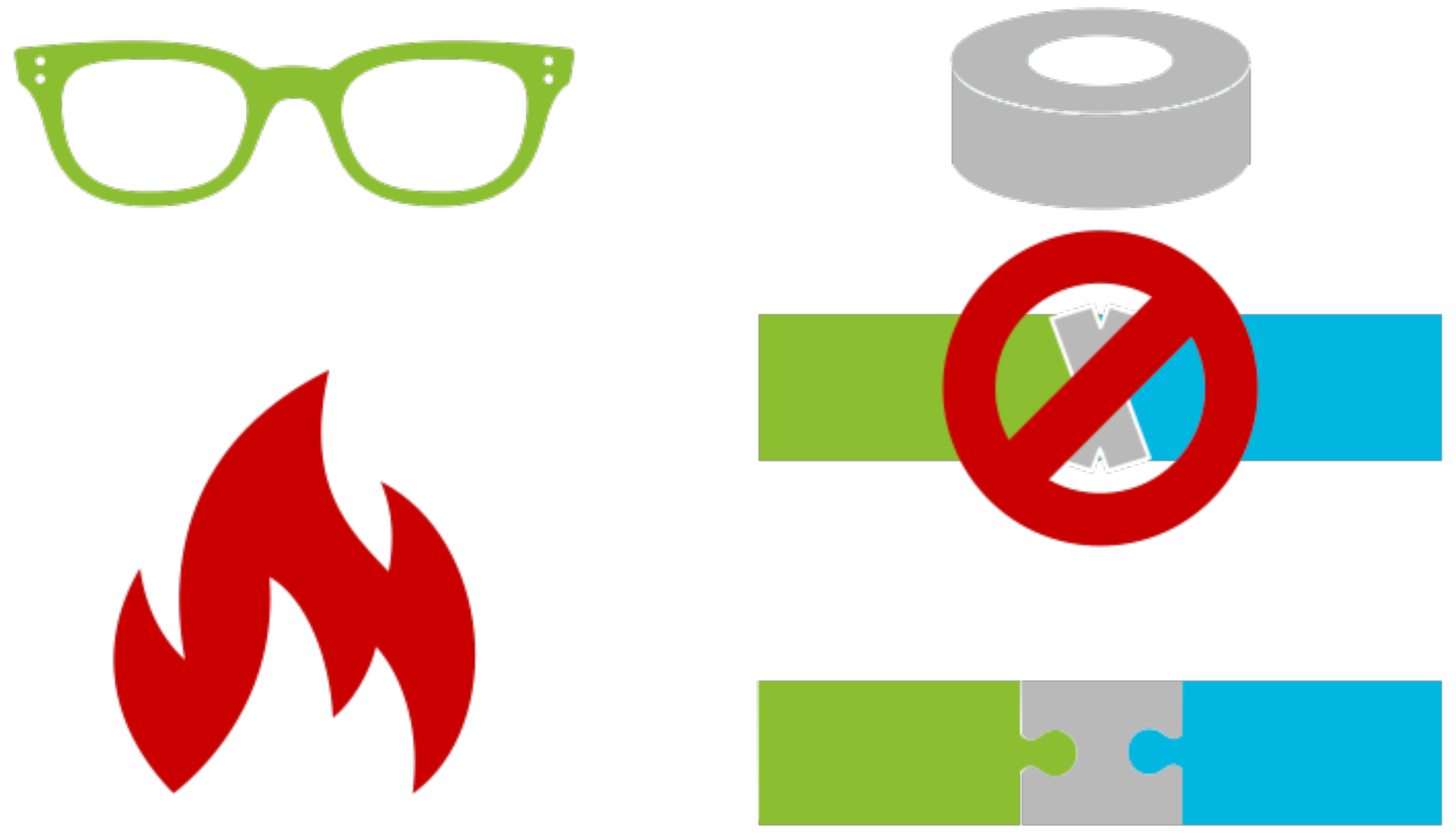


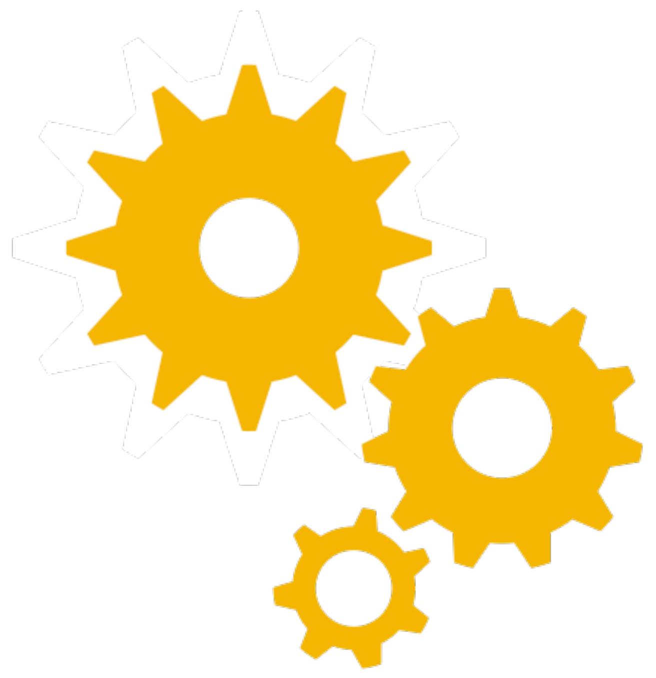
`estimator.fit(df)`



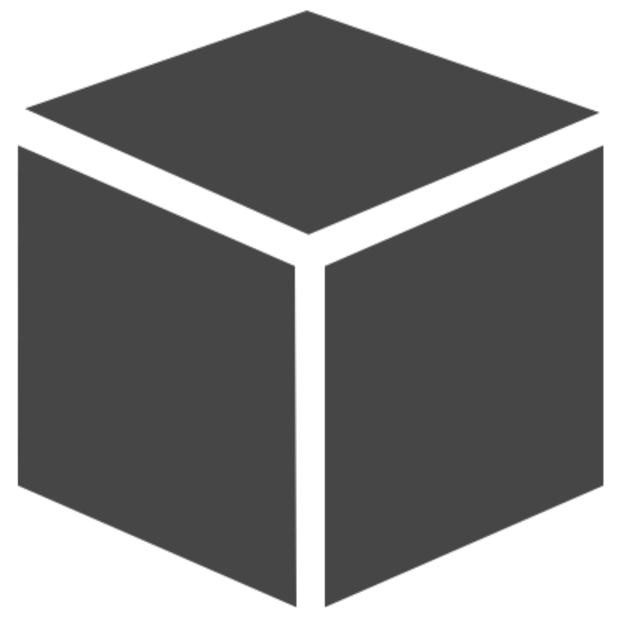
`model.transform(df)`



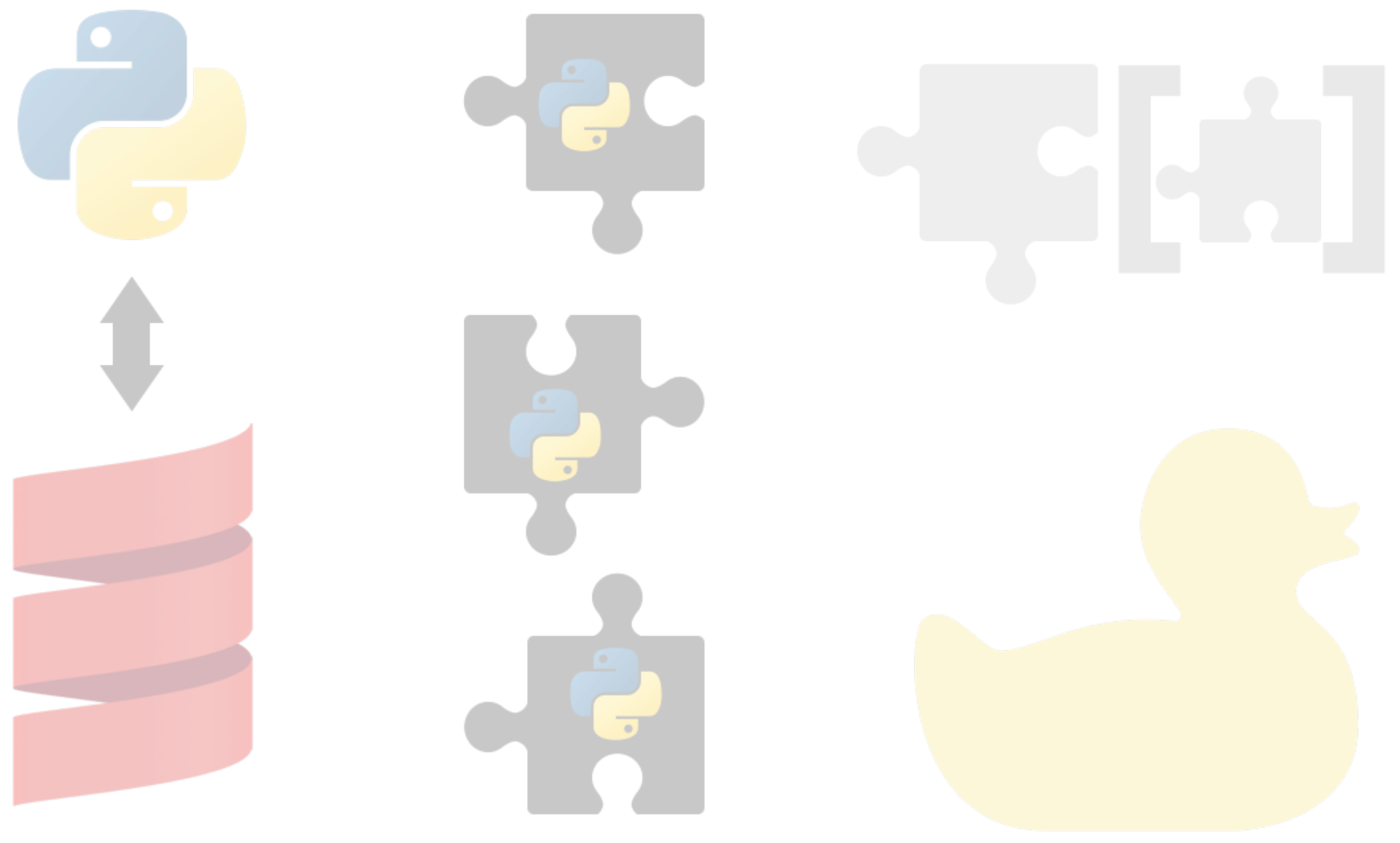


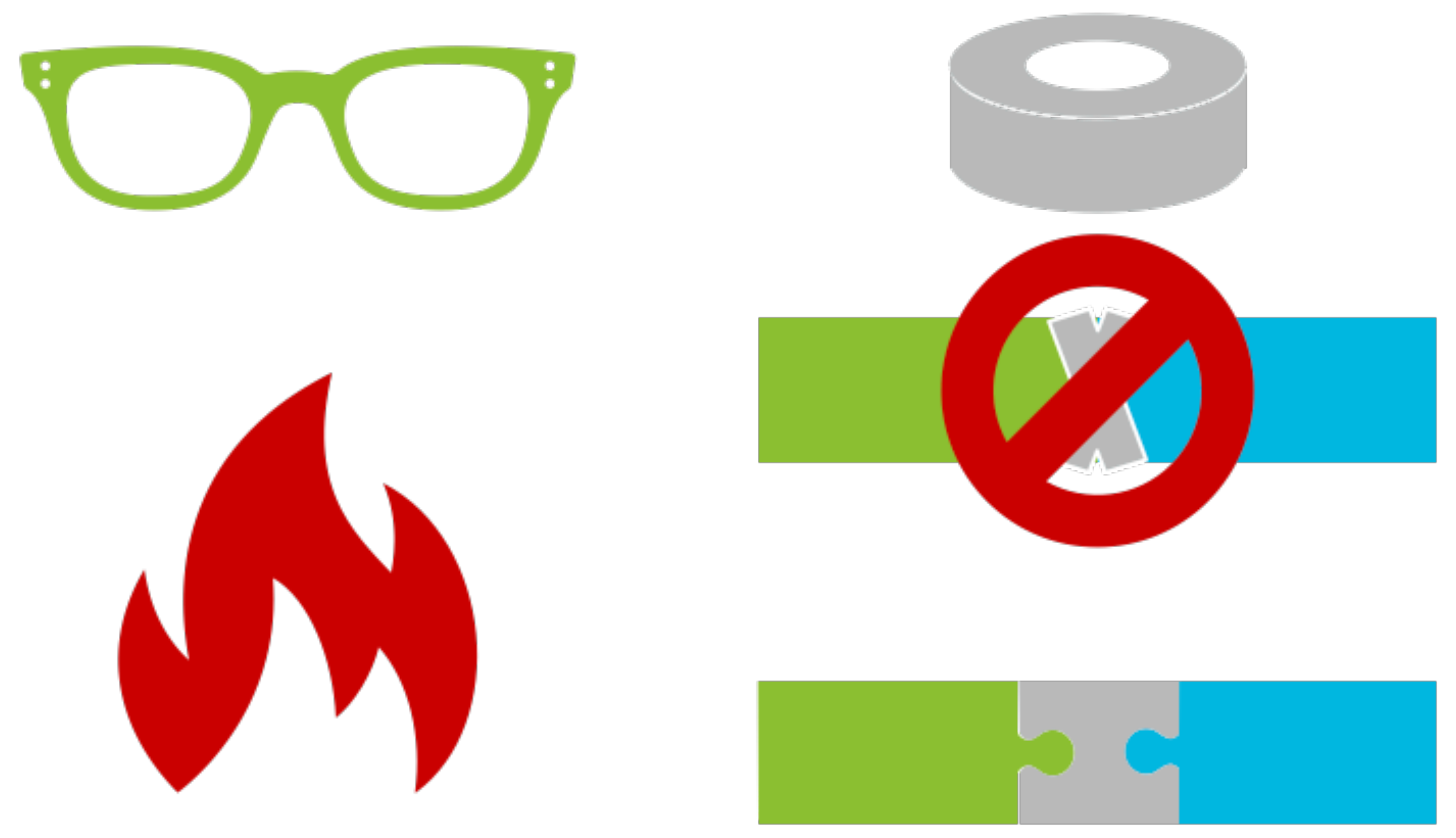


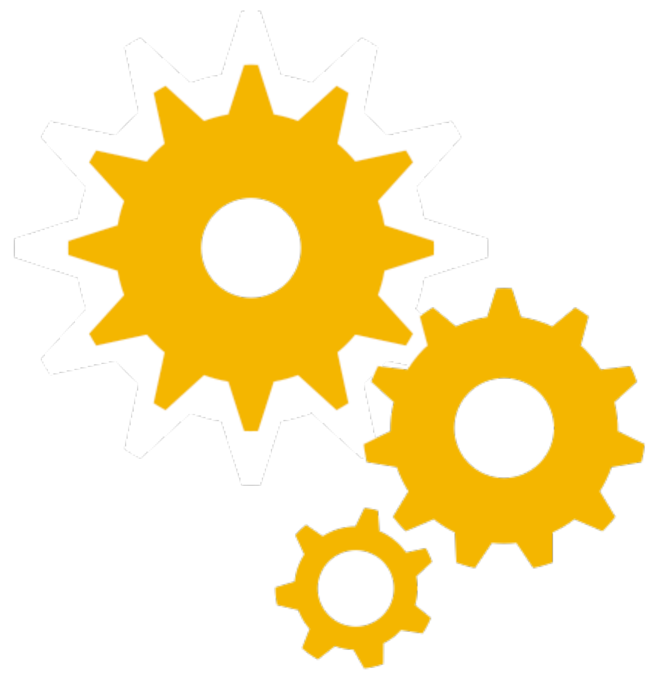
estimator.fit(df)



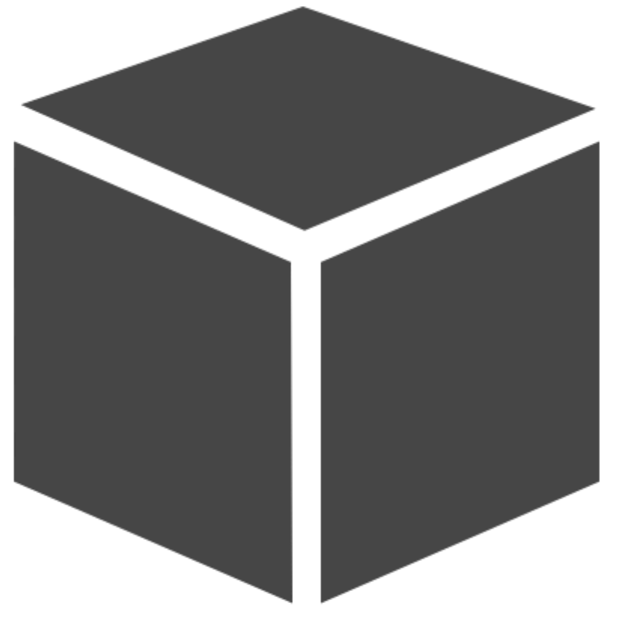
model.transform(df)



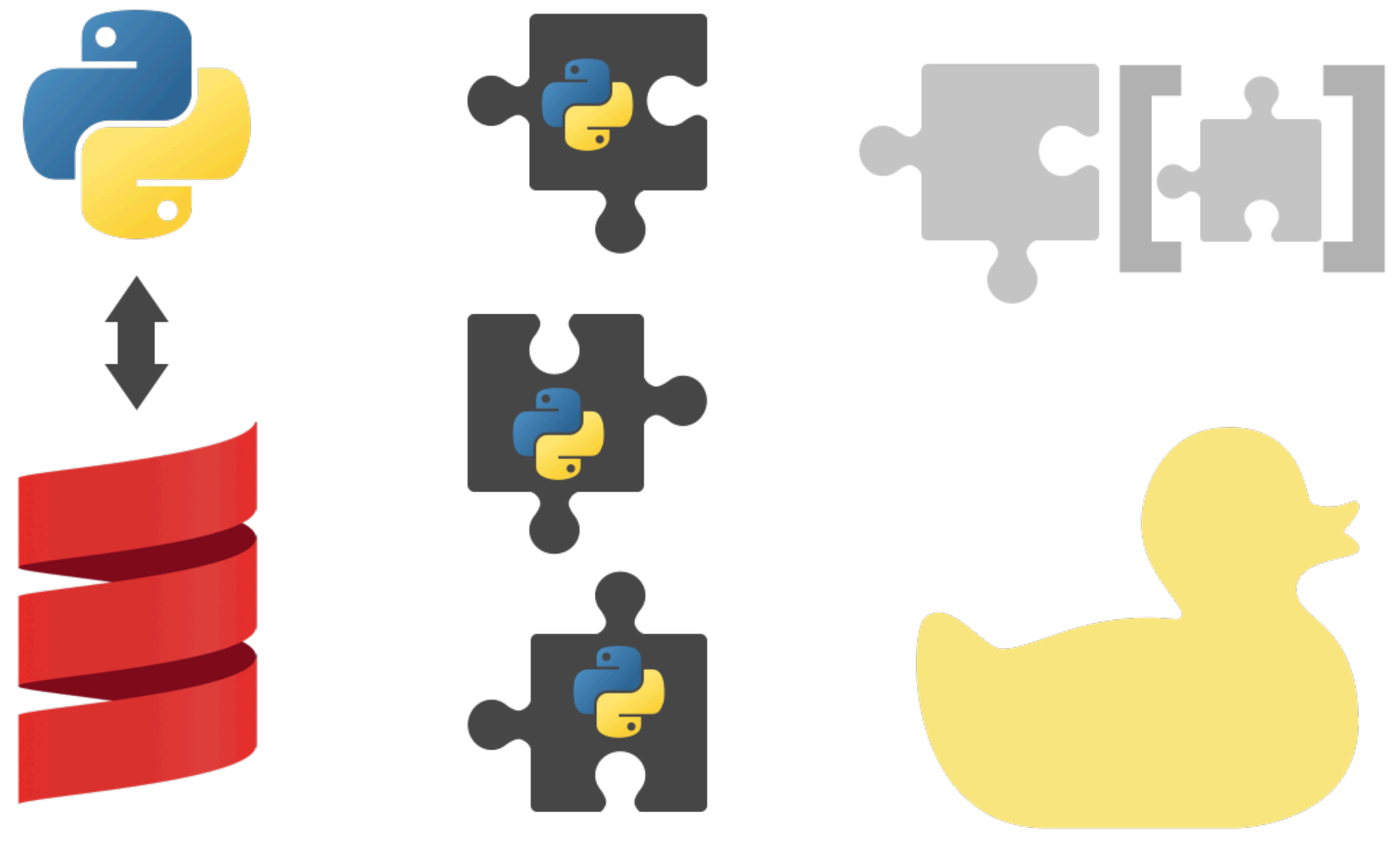


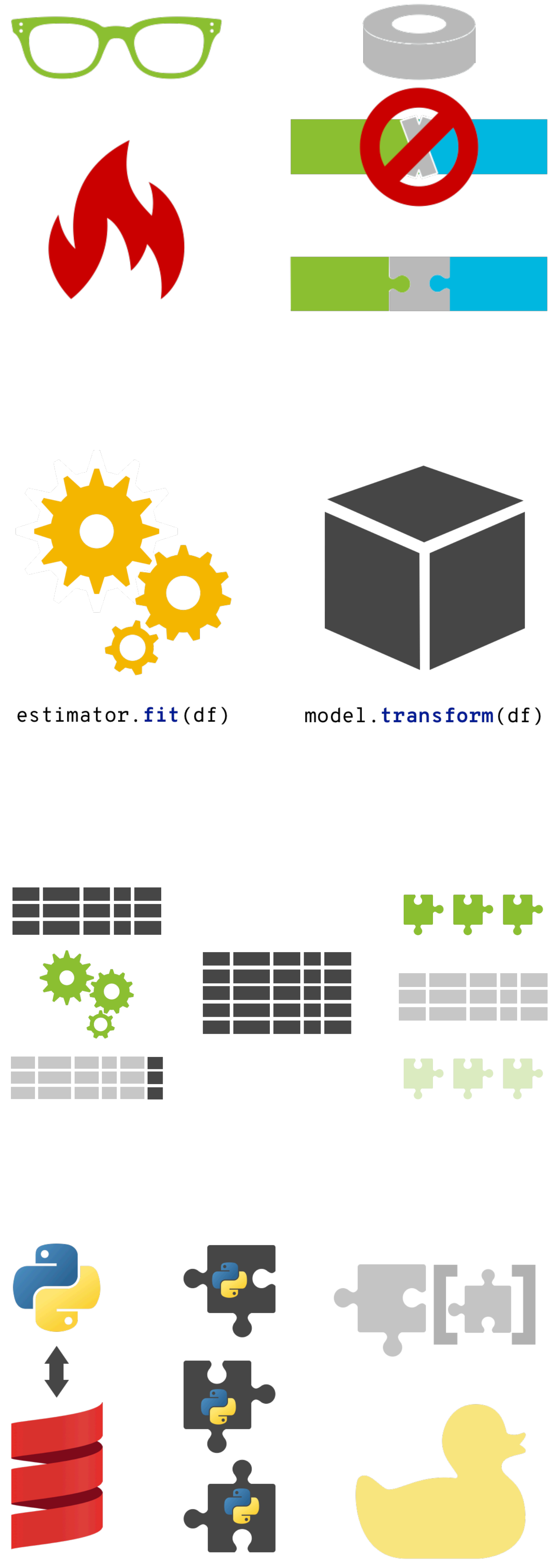


```
estimator.fit(df)
```



```
model.transform(df)
```





KEEP IN TOUCH

https://radanalytics.io

willb@redhat.com • @willb

eje@redhat.com • @manyangled