SPARK+AI SUMMIT EUROPE

Unafraid of change

Optimizing ETL, ML & AI in fast-paced environments

Sim Simeonov, CTO, Swoop

sim at swoop.com / @simeons

#SAISDD13

swoop

INTELLIGENCE BY IPM.ai

omni-channel marketing for your ideal population supported by privacy-preserving petabyte-scale ML/AI over rich data (thousands of columns)

e.g., we improve health outcomes by increasing the diagnosis rate of rare diseases through doctor/patient education

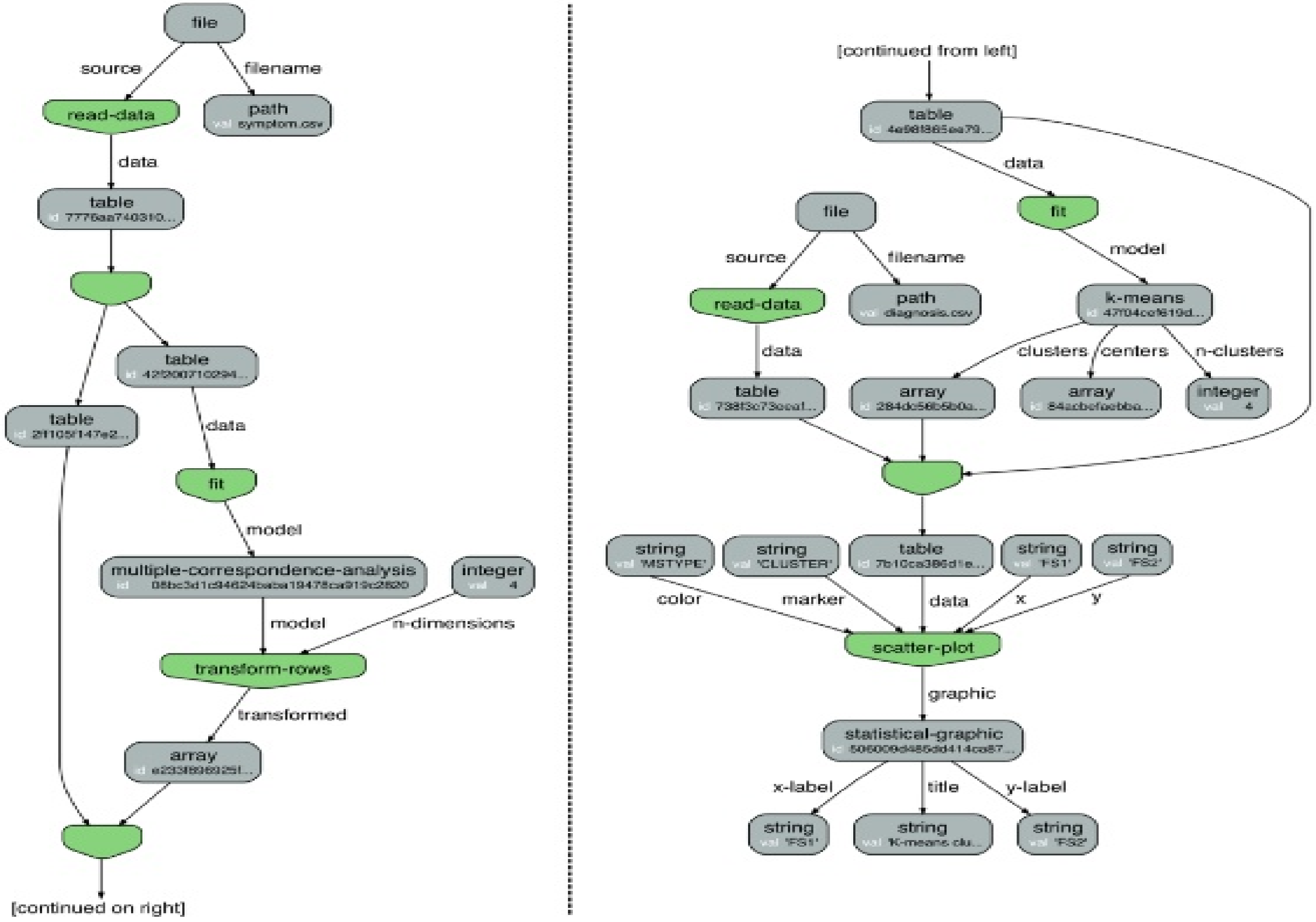
Goal-Based Data Production

Spark-Powered Smart Data Warehouse

Sim Simeonov, Founder & CTO, Swoop

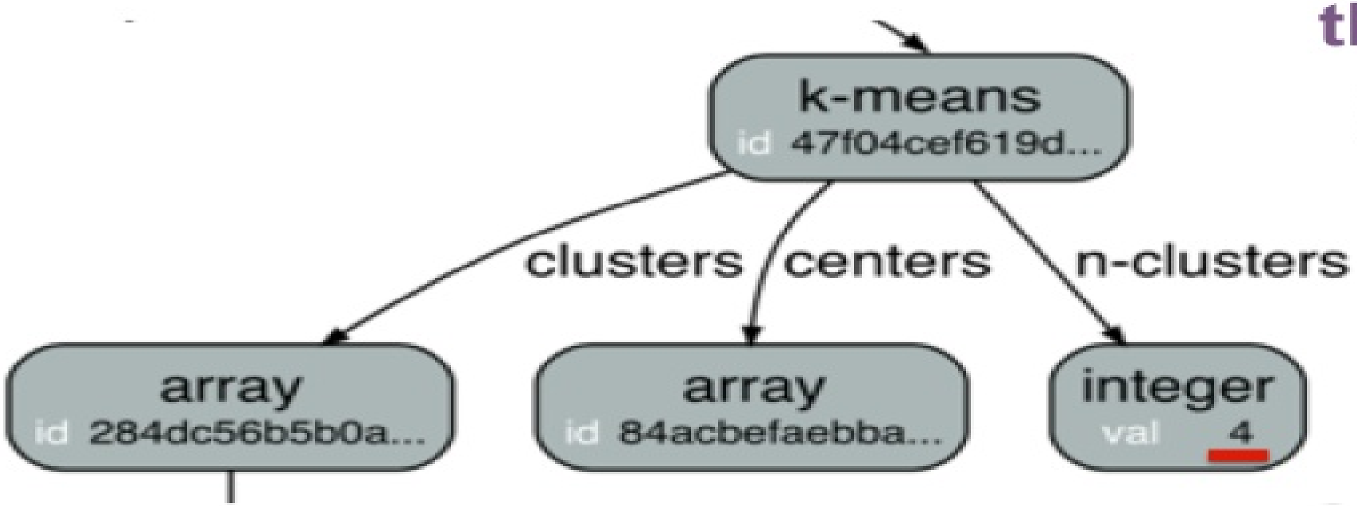
There are two things you don't want to see being made - sausage and ~~legislation.~~
exploratory data science

~ Otto von Bismarck



the myth of data science

The reality of data science is the curse of too many choices (most of which are no good)



- 10? Too slow to train.
- 2? Insufficient differentiation.
- 5? False differentiation.
- 3? Insufficient differentiation.
- 4? Looks OK to me. Ship it!

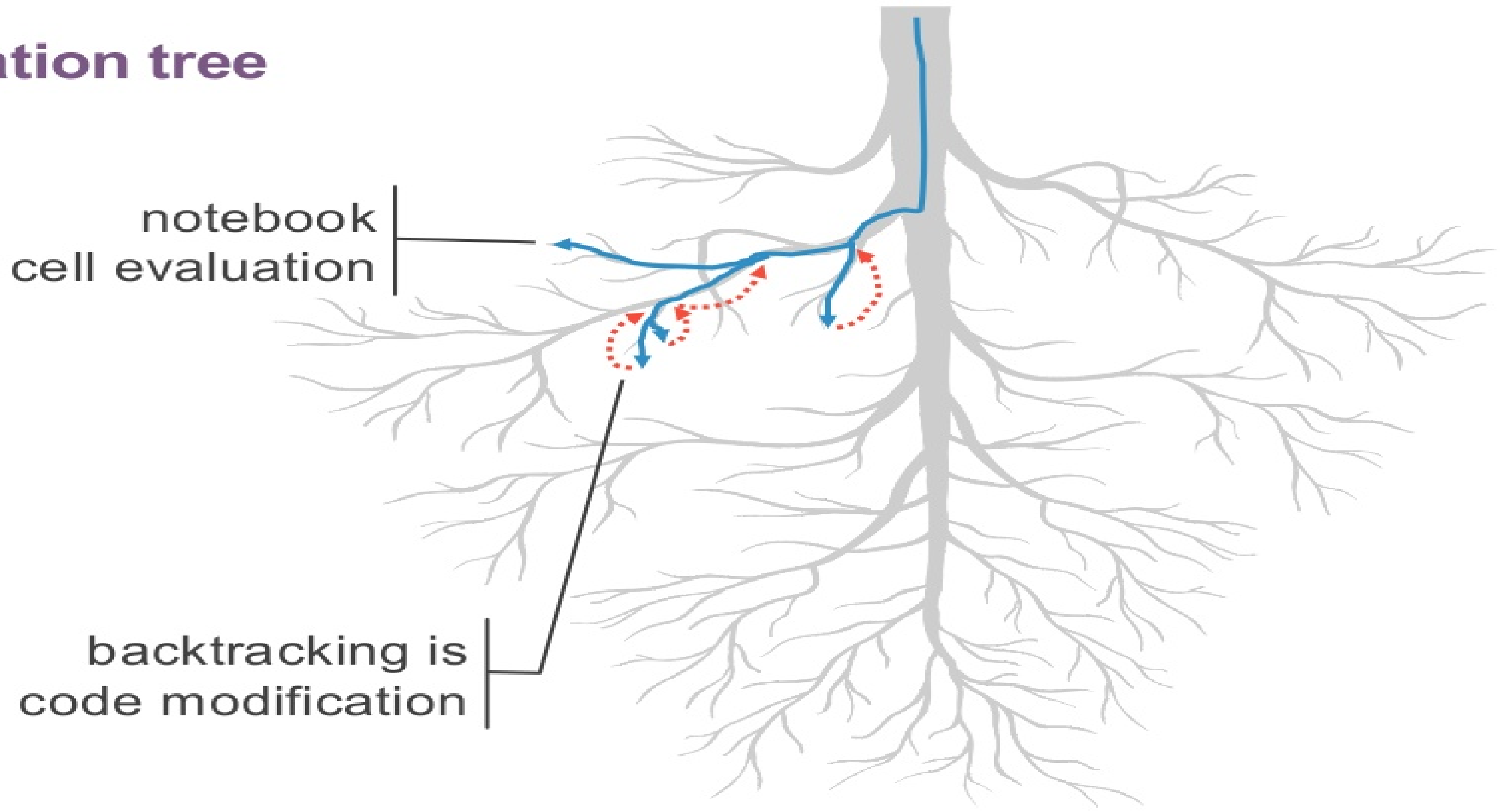
exploratory data science is an

interactive search process

(with fuzzy termination criteria)

driven by the discovery of new information

Exploration tree



Backtracking changes production plans

```
df.sample(0.1, seed = 0).select('x)

df.where('x.isNotNull).sample(0.1, seed = 0).select('x, 'y)

df.where('x > 0).sample(0.1, seed = 0).select('x, 'y)

df.where('x > 0).sample(0.2, seed = 0).select('x, 'y)

df.where('x > 0).sample(0.2, seed = 123).select('x, 'y)
```

Backtracking is not easy with Spark & notebooks

Go back & change code

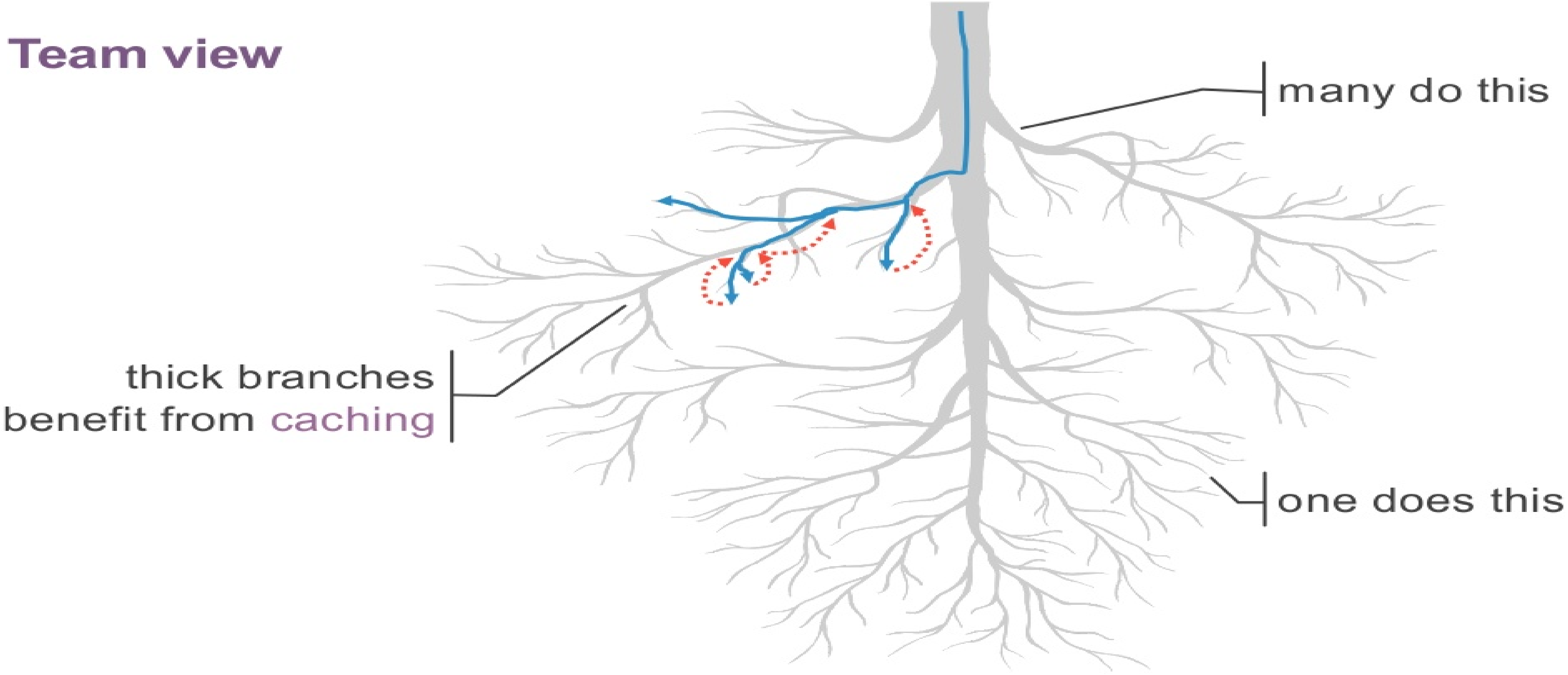
- Past state is lost

Duplicate code

```
val df1 = ...
val df2 = ...
val df3 = ...

// No, extensibility
// through functions
// doesn't work
```

Team view


many do this
thick branches
benefit from caching
one does this

Spark's df.cache() has many limitations

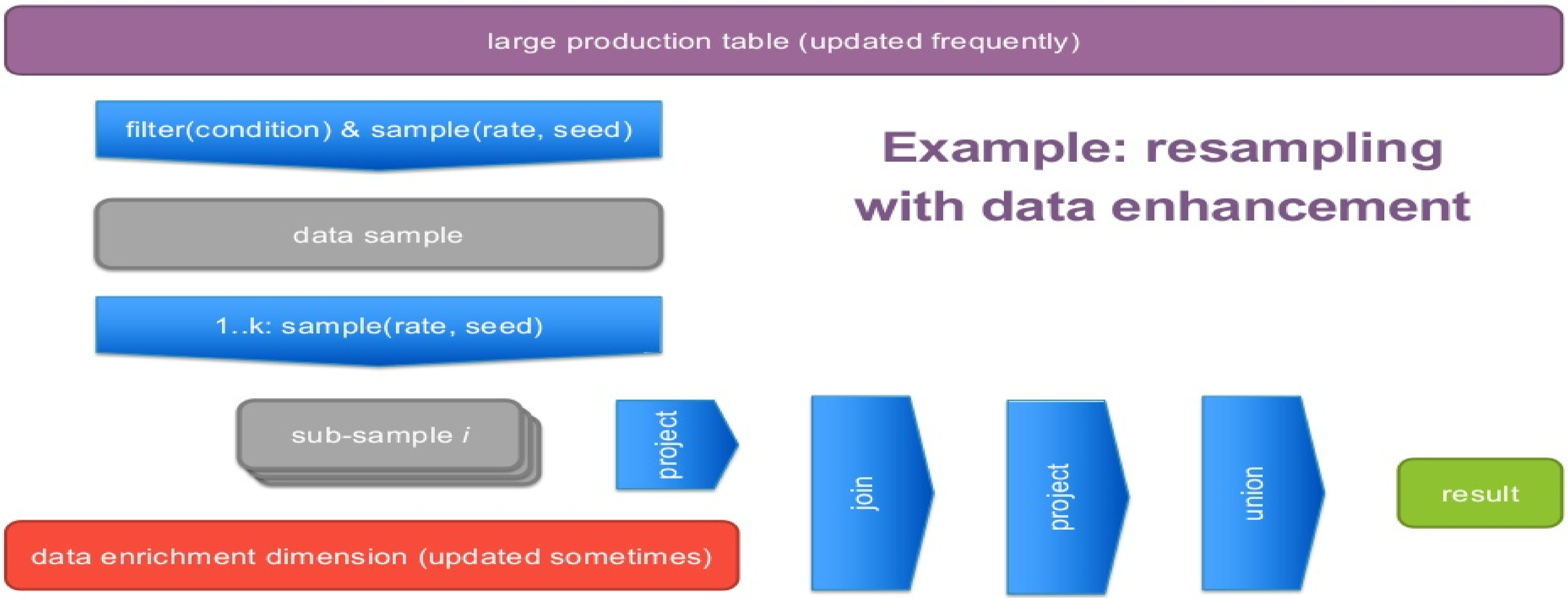
- Single cluster only
  - cached data cannot be shared by the team
- Lost after cluster crash or restart
  - hours of work gone due to OOM or spot instance churn
- Requires df.unpersist() to release resources
  - but backtracking means df (the cached plan) is lost
- No dependency checks
  - inconsistent computation across team; stale data inputs

Ideal exploratory data production requires...

- Automatic cross-cluster reuse based on the Spark plan
  – Configuration-addressed production (CAP)
- Not having to worry about saving/updating/deleting
  – Automated lifecycle management (ALM)
- Easy control over the freshness/staleness of data used
  – Just-in-time dependency resolution (JDR)

```
df.~~cache()~~
df.capCache()
```

part of https://github.com/swoop-inc/spark-alchemy


large production table (updated frequently)
filter(condition) & sample(rate, seed)
Example: resampling with data enhancement
data sample
1..k: sample(rate, seed)
sub-sample *i*
project
join
project
union
result
data enrichment dimension (updated sometimes)

Are primes % 10,000 interesting numbers?

n	note
0	is the additive identity.
1	is the multiplicative identity.
2	is the only even prime.
3	is the number of spatial dimensions we live in.
4	is the smallest number of colors sufficient to color all planar maps.
5	is the number of Platonic solids.
6	is the smallest perfect number.

```scala
// Get the data
val inum = spark.table("interesting_numbers")
val primes = spark.table("primes")
val limit = 10000

// Sample the primes
val sample = primes.sample(0.1, seed = 0)

// Resample, calculate stats & union
val stats = (1 to 30).map { i =>
  sample.select((col("prime") % limit).as("n"))
    .sample(0.2, seed = i).distinct()
    .join(inum.select(col("n")), Seq("n"))
    .select(lit(i).as("sample"), count(col("*")).as("cnt_ns"))
}.reduce(_ union _)

// Show distribution of the percentage of interesting prime modulos
display(stats.select(('cnt_ns * 100 / limit).as("pct_ns_of_limit")))
```

```scala
// Get the data
val inum = spark.table("interesting_numbers")
val primes = spark.table("primes")
val limit = 10000
// Sample the primes
val sample = primes.sample(0.1, seed = 0).capCache()
// Resample, calculate stats & union
val stats = (1 to 30).map { i =>
  sample.select((col("prime") % limit).as("n"))
    .sample(0.2, seed = i).distinct()
    .join(inum.select(col("n")), Seq("n"))
    .select(lit(i).as("sample"), count(col("*")).as("cnt_ns")).capCache()
}.reduce(_ union _)
// Show distribution of the percentage of interesting prime modulos
display(stats.select(('cnt_ns * 100 / limit).as("pct_ns_of_limit")))
```

behind the curtain...

primes.sample(0.1, seed=0).explain(true)

```
== Analyzed Logical Plan ==
ordinal: int, prime: int, delta: int
Sample 0.0, 0.1, false, 0
+- SubqueryAlias primes
   +- Relation[ordinal#2034,prime#2035,delta#2036] parquet

== Optimized Logical Plan ==
Sample 0.0, 0.1, false, 0
+- Relation[ordinal#2034,prime#2035,delta#2036] parquet

== Physical Plan ==
*(1) Sample 0.0, 0.1, false, 0
+- *(1) FileScan parquet default.primes[ordinal#2034,prime#2035,delta#2036]
   Format: Parquet, Location: InMemoryFileIndex[dbfs:/user/hive/warehouse/primes]
```

```
ordinal: int, prime: int, delta: int
Sample 0.0, 0.1, false, 0
  +- Relation[ordinal#2034,prime#2035,delta#2036] parquet
     +- InMemoryFileIndex[dbfs:/user/hive/warehouse/primes]
```

↓ cross-cluster canonicalization

```
ordinal: int, prime: int, delta: int
Sample 0.0, 0.1, false, 0
  +- Relation[ordinal#0,prime#1,delta#2] parquet
     +- InMemoryFileIndex[dbfs:/user/hive/warehouse/primes]
```

↓

CAP (hash: 4bb0070bd5f50bec95dd95f76130f55cd2d0c6bc)

ALM (createdAt: {now}, expiresAt: {based on TTL})

JDR (dependencies: ["table:default.primes"])

Spark's user-level data reading/writing APIs
are inconsistent, inflexible and not extensible.
To implement CAP, we had to design a parallel set of APIs.

Reading

- Inconsistent

```
spark.table(tableName)
spark.read + configuration + .load(path)
                             .load(path1, path2, …)
                             .jdbc(…)
```

- Consistent

```
spark.reader + configuration + .read()
```

Writing

- Inconsistent

```
df.write + configuration + .save()
                           .saveAsTable(tableName)
                           .jdbc(…)
```

- Consistent

```
df.writer + configuration + .write()
// You can read() from the result of write()
```

Spark has two implicit mechanisms
to manage where data is written & read from:
the Hive metastore & “you’re on your own”.
We made **authorities** explicit first class objects.

`df.capCache()` is

```
df.writer.authority("CAP").readOrCreate()
```

Spark offers no ability to inject behaviors into reading/writing.

We added reading & writing **interceptors**.

Dependency management works via an interceptor.

```
trait InterceptorSupport[In, Out] {
  def intercept(f: In => Out): (In => Out)
}
```

It’s not hard to fix the user-level reading/writing APIs.
Let’s not waste the chance to do it for Spark 3.0.

Data science productivity matters.

https://github.com/swoop-inc/spark-alchemy

Interested in challenging data engineering, ML & AI on big data?
I'd love to hear from you. sim at swoop.com / @simeons