Distributed Data Management

Spark Programming Hands-On

IND Discovery Task

Group: Robert'); DROP TABLE Students;-- (Joan Bruguera & Tom Braun)

Problem statement

Given a data set consisting of multiple relational tables, each having multiple columns, we wish to find all distinct pairs of columns where all the values in the first column are included in the second column.

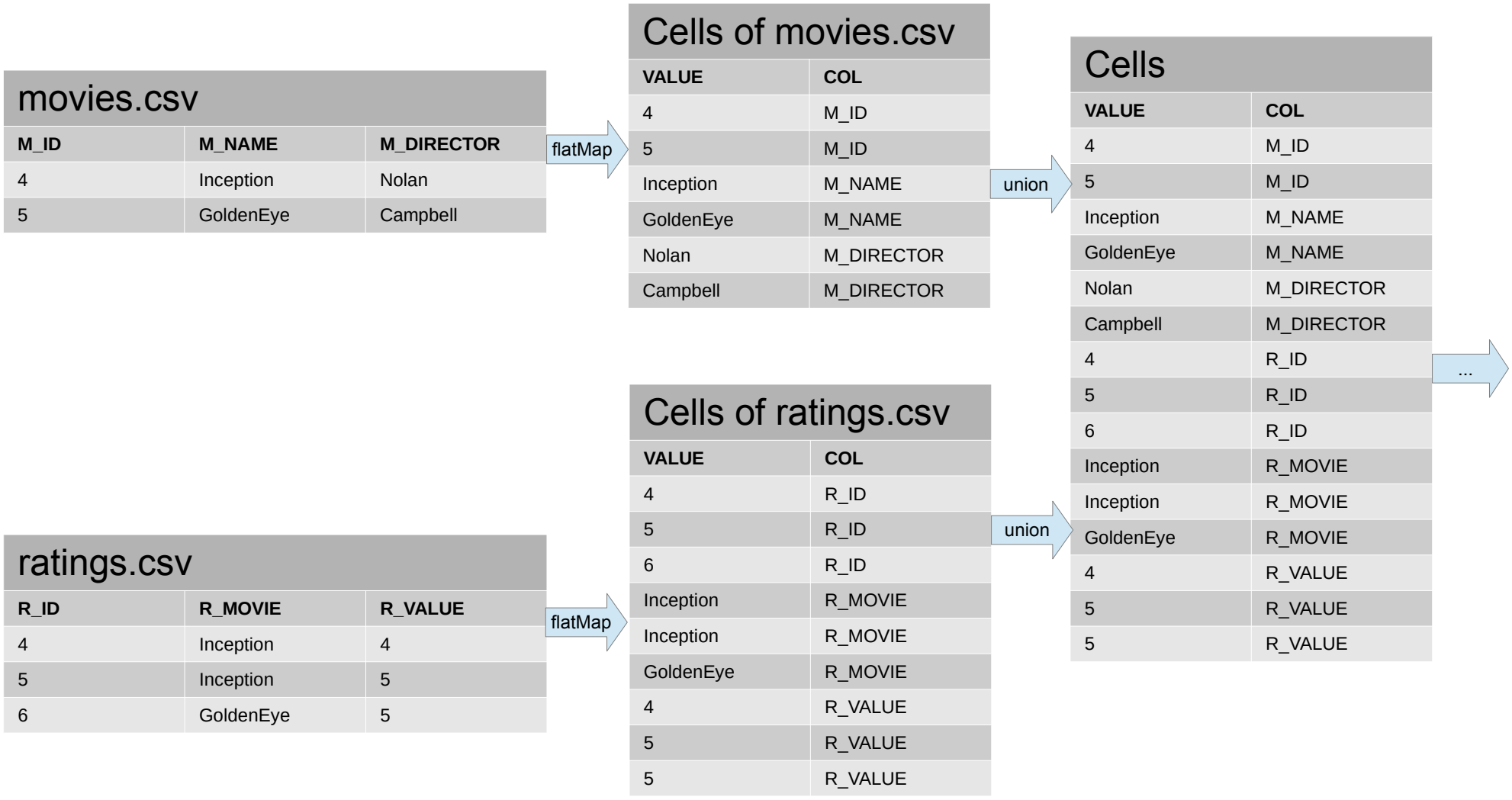
This problem is called Inclusion Dependency Discovery (IND Discovery) and has multiple applications, such as e.g. finding potential foreign key candidates in data dumps where an schema is not available.

Our solution

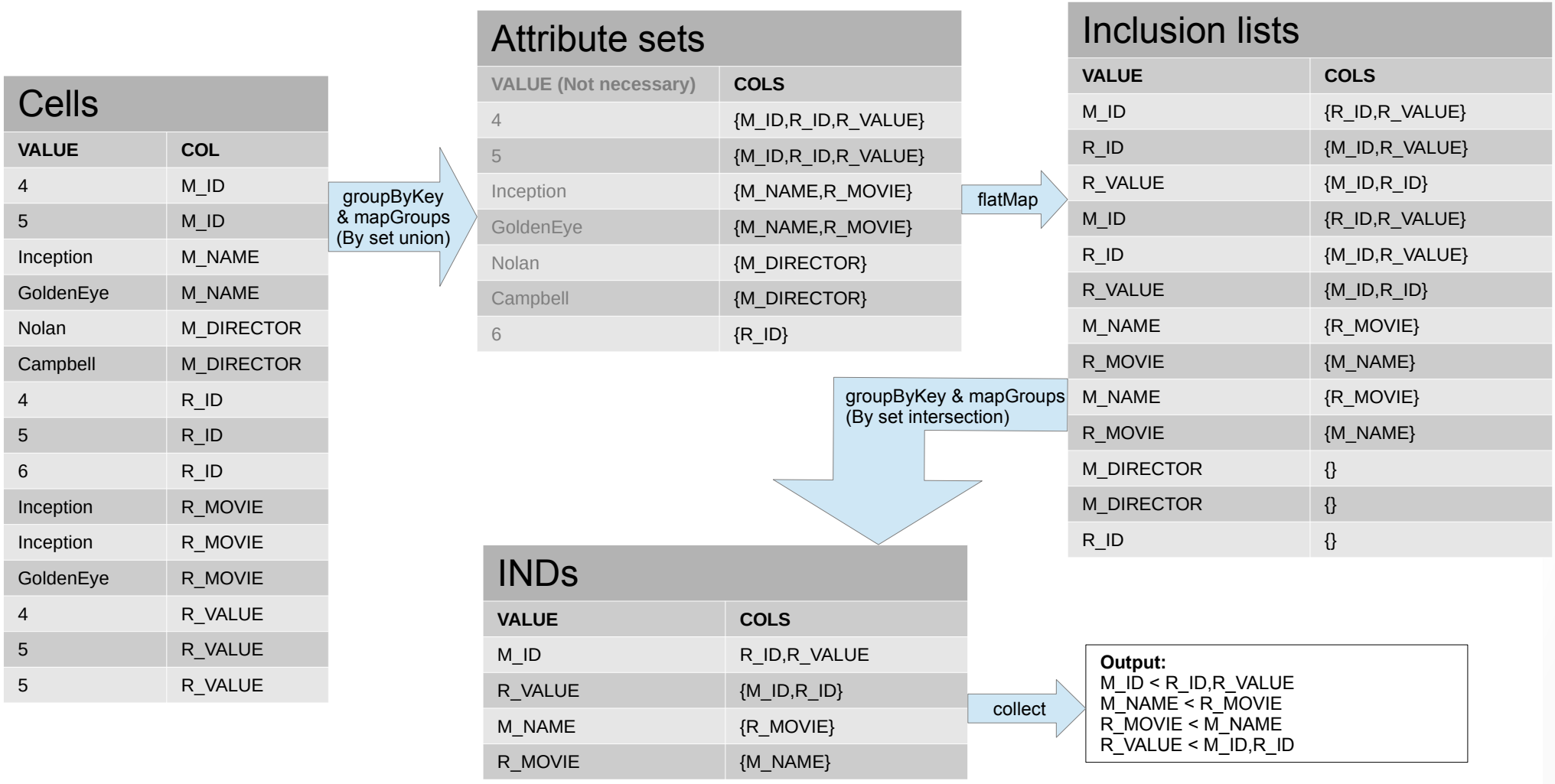
We did a fairly straightforward implementation of the Sindy algorithm[1] in Apache Spark. The advantage of both the Sindy algorithm and its implementation in Spark is that the computation can be easily distributed to multiple nodes and executed in parallel.

[1]: Kruse, S., Papenbrock, T., & Naumann, F. (2015). Scaling out the discovery of inclusion dependencies. Datenbanksysteme für Business, Technologie und Web (BTW 2015).

Spark Pipeline (1): Data input



Spark Pipeline (2): Data transform.



Simplifed code for the pipeline

Data input + Conversion to cells:

```
val cellsByTable = inputs.map(inputName => {
  val table = spark.read.[...]..csv(inputName)
  val columns = table.columns
  table.flatMap(r => (0 until r.length).map(i => (r.getString(i), columns(i))))
})
```

Union of all cells:

```
val allCells = cellsByTable.reduce((a,b) => a.union(b))
```

Compute Attribute Lists from cells:

```
val attributeSets = allCells.groupByKey { case (value, _) => value }
  .mapGroups { case (_, iterator) => iterator.map(x => x._2).toSet }
```

Compute Inclusion Lists from Attribute Lists:

```
val inclusionLists = attributeSets.flatMap(r => r.map(v => (v, r – v)))
```

Compute INDs from Inclusion Lists:

```
val inds = inclusionLists.groupByKey { case(k, _) => k }
    .mapGroups { case (k, sets) => (k, sets.map(a => a._2).reduce((a, b) => a.intersect(b))) }
    .filter(r => r._2.nonEmpty)
```

Collect and print INDs:

```
inds.collect().sortBy(x => x._1).foreach(x => println(x._1 + " < " + x._2.mkString(", ")))
```

Optimizations: Unique/repeated

We optimized the naive Sindy implementation based on the following observation:

In typical data sets, many cell values are either **unique** (they appear only once in the entire dataset) or **repeated** (they appear many times in the dataset, often within the same column)

Examples of an unique cell: User-written comment, UUID
Examples of a repeated cell: Rating value (0-10), boolean (true/false)

Optimizations: Unique/repeated

To minimize the impact of **repeated cells**: We run distinct() on the cells of each CSV (RDD "Cells of xyz.csv" in the diagram).

To minimize the impact of **unique cells**: We run distinct() on the collection of attribute sets (RDD "Attribute sets" in the diagram).

Both transformations are correct: The resulting INDs are the same (without further changes to the algorithm)

Optimizations: Set vs bit set

In our implementation we work with `DataSets`, and we use Scala's `Sets` in various steps of the pipeline, for union and intersection operations.

We tried to replace the usage of Scala's `Sets` with more lightweight bit sets, in order to reduce the size of the objects in the pipeline.

First attempt: Use one of the bit set classes available on the Java or Scala standard library.
Problem: None of those classes have an Encoder for Spark, so we can't use them as values in the `DataSet`.

Second attempt: Work with a plain `long` and bitwise operations.
Result: Works correctly, but the performance improvement was small (<10%) and limited our implementation to max. 64 columns

As a result, we kept the original implementation with `Sets`.

Results

On a typical dual-core laptop, **we were able to process the given sample TPCH data set** (size: ~400MB) can be processed **in about 40 seconds.**

This is fast enough for this data set, but **the numbers are not that great**: That amounts to just 10MB/s!

Most likely causes:

- Our sample data set is too small and much of that time is just initialization overhead
- Spark being optimized for massive distributed instead of local execution

To properly evaluate our implementation we'd need to run the algorithm on a compute cluster and with a bigger data set, but unfortunately we didn't have the resources and time to do so.