TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Dr. Thomas Neumann

Exercises for *Foundations in Data Engineering*, WiSe 23/24

Alexander Beischl, Maximilian Reif (i3fde@in.tum.de)
http://db.in.tum.de/teaching/ws2324/foundationsde

Sheet Nr. 09

Exercise 1 Order the following workloads by how well they are suited for cluster computing. Assume a cluster of machines which are connected via ethernet and that the dataset is distributed onto the machines when the workload starts. A workload that is well suited for cluster computing decreases in execution time proportional to the number of machines in the cluster.

- Search in text documents
- The toy example from the lecture (Basic Building Blocks, slide 26-35)
- Sorting records
- Breadth-first search
- Join of two relations (both relations to big for one machine)
- Shuffling a dataset

Solution:
From well suited to hard to distribute:

- *Search in text documents*, *The toy example from the lecture*: Operation on local fragment of the data. Only result has to be merged.
- *Shuffling a dataset*: Apply shuffling schema on each local dataset. Redistribute data according to shuffling schema.
- *Sorting records*: Apply local sorting first, afterwards distribute local fragments and merge them.
- *Join of two relations*: Easy if one relation is small (and can be distributed). Hard if both relations are big - high communication cost. Slides (Advanced SQL, Slides 54-58)
- *Breadth-first search*: Hard. Entire dataset needs to be considered for every iteration and results must be exchanged.

Order is generalized and can be different depending on the concrete use-case and the applied algorithms, e.g., NeoJojn. The idea of the task is to discuss the challenges of distributed computing.

Exercise 2

Formulate map-reduce programs to handle the following tasks:

1. For the multi-sets $X : [(a, b)], Y : [(c, d)]$ find all combinations where $b = c$.

Solution:

```
mapX((a,b))
  emit(b, ('X',a,b))

mapY((c,d))
  emit(c, ('Y', c, d))

reduce(k, l)
  lX = l.filter(_1 == 'X')  // _1 gets the first element of a tuple
  lY = l.filter(_1 == 'Y')
  for x in cross(lX,lY)
    emit("result", x)

main(X,Y)
  mX = mapAll(mapX, X)
  mY = mapAll(mapY, Y)
  return reduceAll(reduce, mX + mY)["result"]
// Note: We have to get the singleton result using '...["result"]'.
```

2. For the documents $D : [(name, [w])]$ (where D is a list of documents, in which each document is a list of words w), find the words which all documents have in common.

   Solution:

```
//count docs
mapCount((name, words : list))
  emit('count', 1)

getCount(k, l : list)
  emit(k, sum(l))

mapDocuments((name, words : list))
  for word in unique(words):
    emit(word, 1)

reduceWords(word, counts : list)
  emit(word, sum(counts))

main(D)
  c = mapAll(mapCount, D)
  numDocs = reduceAll(getCount, c)['count']
  w = mapAll(mapDocuments, D)
  wordsWithDocCount = reduceAll(reduceWords, w)
  return wordsWithDocCount.filter((w, cnt) → cnt == numDocs)
```

3. Compute AB for the two matrices A and B.

Solution:
Dimensions $A : n \times m$, $B : m \times p$
Matrices represented as $(index\_row, index\_column, value)$.

```
mapA((in_row, in_col, value))
  for out_col in 1 to p
    emit((in_row, out_col, in_col), value)

mapB((in_row, in_col, value))
  for out_row in 1 to n
    emit((out_row, in_col, in_row), value)

reduceMult((out_row, out_col, idx), values)
  emit((out_row, out_col), prod(values))

reduceSum((out_col, out_row), values)
  emit((out_row, out_col), sum(values))

main(A,B)
  mapAll(mapA, A)
  mapAll(mapB, B)
  reduceAll(reduceMult)
  reduceAll(reduceSum)
```

Exercise 3 This exercise is about getting familiar with the Spark Dataset API.

To get started, open a spark shell and load the *song dataset* into a dataset. The data is an excerpt from the *Million Song Dataset*. Make sure that appropriate data types are used for each column. You can check the types e.g. with the `printSchema` function.

For a list of functions offered by the dataset API please refer to the *dataset documentation*. To get started with importing a csv file, have a look at *this tutorial*.

Once the dataset is loaded, use the API to answer the following questions:

1. List all songs from the year 2000.
2. In which year is the first song from this dataset published? (For some songs in the dataset, the year is set to 0. In this case, the year is unknown.)
3. Which artist is the hottest, according to the `artist_hotttnesss` column?
4. Which album has most songs on it? (Use release text and artist to identify album.)
5. Find song pairs with equally familiar artists. That is: `a.artist_familiarity = b.artist_familiarity` How many pairs are there?
6. Find song pairs with similarly familiar artists.
   That is: $|\texttt{a.artist\_familiarity} - \texttt{b.artist\_familiarity}| < 0.001$
   How many pairs are there? You may notice, that when trying to run this query, it does not terminate (in reasonable time). Call `explain` on the final dataset. This will show the execution plan which will be used to retrieve the result. Most likely, the root node of the plan you are seeing is a `CarthesianProduct` operator. It produces a cross product of all its input and applies a filter operation on the result. Thus, it is quite slow for larger amounts of data. Try to reformulate the query so that no `CarthesianProduct` is used.

Solution:

```
val songs = spark.read.format("csv").option("header", true).option("
    inferSchema", "true").load("songs.csv").cache()

// Optional: you can use this time function to measure execution time.
def time[R](block: => R): R = {
  val t0 = System.nanoTime()
  val result = block    // call-by-name
  val t1 = System.nanoTime()
  println("Elapsed time: " + (t1 - t0)/1000000 + "ms")
  result
}

// 1. Get all songs from the year 2000.
songs.filter($"year" === 2000).show()

// 2. In which year is the first song from this dataset published?
songs.filter($"year" > 0).agg(min($"year")).show()

// 3. Which artist is the hottest?
val hottest = songs.agg(max($"artist_hotttnesss")).first().get(0)
songs.filter($"artist_hotttnesss" === hottest).select($"artist_name").
    distinct().show()
songs.filter($"artist_hotttnesss" === hottest).select($"artist_name").
    distinct().explain()

// 4. Which one is the largest album? (Use release text and artist to
    identify album)
songs.groupBy($"release", $"artist_name").count().orderBy($"count".
    desc).show()
songs.groupBy($"release", $"artist_name").count().orderBy($"count".
    desc).explain()
```

```
// 5. Find artist pairs that are equally familiar. That is: artist1.
    familiarity = artist2.familiarity
songs.as("a").join(songs.as("b"), $"a.artist_familiarity" === $"b.
    artist_familiarity").count()

// 6. Find artist pairs that are equally familiar. That is: |artist1.
    familiarity - artist2.familiarity| < 0.001
songs.as("a").join(songs.as("b"), abs($"a.artist_familiarity" - $"b.
    artist_familiarity") < 0.001).explain()
// Does not terminate:
songs.as("a").join(songs.as("b"), abs($"a.artist_familiarity" - $"b.
    artist_familiarity") < 0.001).count()


// Let's create buckets, join on buckets first and then check the
    original constraint again
val songsBuck = songs.withColumn("fam_bucket", floor($"
    artist_familiarity" / 0.001)).cache()
// Create a dataset with rows for neighboring buckets
val songsBuckNeighbors = songsBuck.withColumn("fam_bucket", explode(
    array($"fam_bucket" - 1, $"fam_bucket", $"fam_bucket" + 1)))
// Join on bucket and constraint
songsBuck.as("a").join(songsBuckNeighbors.as("b"), ($"a.fam_bucket"
    === $"b.fam_bucket") && abs($"a.artist_familiarity" - $"b.
    artist_familiarity") < 0.001).count()
```

The solution of the last question is very similar to the taxi bonus project. You may want to point this out to your students.