Group **A**

Please fill in your name and registration number (Matrikelnr.) **immediately**.

Exam 2 ON	**SOLUTION KEY**	07.10.2019
	Advanced Database Systems (**184.780**)	**GROUP A**
Matrikelnr.	Last Name	First Name

Duration: 75 minutes. Provide the solutions on the designated pages. **Good Luck!**

Question 1: (4 credits)

For each of the statements below, decide if it is true or false and tick the corresponding circle. You get +1 credit for each correct answer, −1 credit for each wrong answer and 0 credit if you leave the answer open. In total, you always get ≥ 0 credits on the entire exam question 1.

1. In Spark, the filter operation may be applied to mutable variables but not to immutable variables.
   true ◯ false ⊗
2. In Spark, the user should combine several small computation steps to a single step in order to avoid performance loss.
   true ◯ false ⊗
3. In the MapReduce execution model, if a map worker fails, then all map tasks by this worker have to be redone.
   true ⊗ false ◯
4. In Hive, the user can choose between several execution engines. These choices include MapReduce and Spark.
   true ⊗ false ◯

Question 2: (4 credits)

A colleague of yours, who is writing her/his first MapReduce programme, is asking you for advice: She/he has recently heard about combiners but did not understand what they are good for.

a) [2 credits] Briefly explain to her/him

- when is a combiner applicable,
- what is the goal of a combiner, and
- how is this goal achieved?

- Applicability: computation of an associative and commutative function; combiner and reducer have the same type of input; moreover, the combiner must produce output of the same type as its input.
- Goal: reduce the communication cost
- Method: shift part of the reducer functionality to the mapper nodes; thus, only aggregated data has to be communicated from the mapper nodes to the receiver nodes.

b) [2 credit] The application which your colleague had in mind involves the computation of the average function. Can a combiner be used for this purpose?

- If your answer is yes, give a brief sketch of the combiner.
- If your answer is no, give a short argument, why not?

- mapper: for every number n from the input, produce a key value pair (k, v) where k is always the same (e.g., simply left empty), i.e., we have a single reducer, and $v = (1, n)$.

  The idea is that v is a pair consisting of count and sum.

- combiner/receiver: receives a list of pairs $(c_1, n_1), \ldots, (c_\ell, n_\ell)$ and produces a single key-value pair with unchanged key.

  The value is computed as value = N/C with $N = \sum_{i=1}^{\ell} n_i$ and $C = \sum_{i=1}^{\ell} c_i$.

Question 3: (4 credits)

In this example you are asked to write a given type of join in MapReduce. The implementation details are left open, and there are multiple possible ways to implement this join in MapReduce. You will get full points as long as your solution sketches out a correct MapReduce program of the join type that is defined below.

- The **Left Outer Join** of R and S ($R \bowtie S$), is the result of the (natural) join between R and S, combined with those tuples of R with no join partner, with the missing attributes from S replaced by NULLs:

R

A	B	C
1	2	3
2	2	4

S

B	C	D
2	3	5

R ⋈ S

A	B	C	D
1	2	3	5
2	2	4	NULL

Assume for your solution the given relational schema: $R(A, B, C)$ and $S(B, C, D)$.

In addition to sketching out a MapReduce program, also identify the communication cost of your algorithm, as a function of the input size.

Your Answer:

(Solved using a partition join)

Mapper

For each input tuple (a,b,c) from R emit a tuple $\langle$(b,c), (a,'R')$\rangle$, and for each input (b,c,d) from S emit a tuple $\langle$(b,c), (d,'S')$\rangle$, where 'R' and 'S' mark the origin of this tuple.

Reducer

The input is a key (b,c) and a list of pairs consisting of attribute and origin flag.
Determine - using the origin flags - A, the list of values from R, and D, the list of values from S.

If both are non-empty, then perform a cross-product and emit a tuple for every possible combination.

If A is non-empty, but D is empty, then emit for every a $\in A$ a tuple (a,b,c,NULL).

Otherwise do nothing.

Communication Cost:

$|R| + |S|$

Question 4: (4 credits)

a) Evaluation of Spark/Scala [2 credits]
For this part of the question, you are given some Spark/Scala code and are supposed to evaluate the results. Don't worry about the "syntax" of your output, it is only important that it is clear that you understood the underlying semantics of Spark/Scala.

```
val fact5 = List(1, 2, 6, 24, 120)
val result = fact5.filter(x => x % 2 == 0 && x % 3 == 0).map(x => x/6)
```

```
result = List(1, 4, 20)
```

Note: '%' is the *modulo* operator (i.e., the rest after division).

```
val numbers = List(2, 4, 4, 5, 7, 7, 8, 10, 14)
def foo(num:Int) = List(num, num*2, num+3)
val otherNumbers = List(2, 4, 7).flatMap(foo)
val result = numbers.sum - otherNumbers.sum
```

```
result = 0
```

Note: '`.sum`' produces the sum of a list of integers

b) Writing Spark/Scala functions [2 credits]
You are given a list of random words as input, for instance:
`val words = List("madam", "letter", "rotator", "mammal")`
A *palindrome* is defined as a word which is the same when read from left to right, or right to left. Examples: "racecar", "radar" or "level". For each word in your input, check if it's a palindrome, if not, add the warning ": no palindrome!" to the string. You can change the order of the list of words, but each word from the input must be present (as a prefix at least) in your output.

```
Many solutions:
val palindromes = for ( w <- words) yield {if (w.reverse == w) {w} else {w + ": no palindrome!"}}
Alternatively:
val palindromes = words.filter(x => x.reverse == x) ::: words.filter(x => x.reverse != x).map(x
=> x + ": no palindrome!")
```

Note: You can use the '`.reverse`' method to reverse the letters of any string. *You are expected to write a programmatic solution, writing down a static list of strings will get 0 credits.*

Question 5: (4 credits)

You are given a database for a retailer.

Note: underlines represent primary keys, *italics* represent foreign keys. We don't state the type of attributes here, just assume some reasonable defaults (string, int, float, ...)

manufacturer	(name, country. ceo, main_product: *product.id*)
product	(id,name, first_produced, made_by: *manufacturer.name*)
offers	(id: *product.id*, brand: *retailer.brand*, price, created_on)
retailer	(brand,country, no_of_stores)
discount	(by_brand: *retailer.brand*, code, for_product: *product.id*)

Assume the dataset is loaded and the corresponding views have already been created in Spark SQL. For the dataframes, simply use the name of each table appended with "DF" (e.g.: productDF, discountDF, etc.). You don't need to generate these dataframes, just assume they are already loaded and usable.

You are now given a number of queries, either in Spark SQL or in the Dataframe API. Your task will be to state an equivalent query of the other type, i.e. give a Spark SQL equivalent if the Dataframe API version is given, or vice-versa.

```
val query1DF = manufacturerDF.join(retailerDF,manufacturerDF("country") === retailerDF("country"))
                             .groupBy(manufacturerDF("name"))
                             .agg(count(retailerDF("brand")))
```

```
val query1SQL = spark.sql("SELECT name, COUNT(brand) FROM manufacturer NATURAL JO
IN retailer GROUP BY name"
```

```
val query2DF = retailerDF.join(offerDF,offerDF("brand") === retailerDF("brand"))
                         .groupBy(retailerDF("brand"))
                         .agg(max("price").as("maxPrice"))
                         .select(retailerDF("brand"), $"maxPrice" + 100 )
```

```
val query2SQL = spark.sql("SELECT r.brand, MAX(price)+100 FROM retailer r NATURAL
JOIN offers o GROUP BY r.brand")
```

```
val query3SQL = spark.sql("SELECT name, id
                           FROM manufacturer m, product p
                           WHERE p.made_by = m.name
                           AND NOT EXISTS (SELECT *
                                           FROM discount d
                                           WHERE d.for_product = p.id) ")
```

```
val query3DF = manufacturerDF.join(productDF,productDF("made_by") === manufacturerDF("name")).select(manufacturerDF("name"),productDF("name")).except(manufacturerDF.join(productDF,productDF("made_by") === manufacturerDF("name")).join(discountDF,discountDF("for_product") === productDF("id")).select(manufacturerDF("name"),productDF("name")))
```

Note: You can use ".except" to form the difference of two dataframes **of the same schema**.

```
val query4SQL = spark.sql("SELECT    p.name,r.brand, AVG(o.price)
                           FROM      product p, retailer r, offers o
                           WHERE     p.id = o.id AND r.brand = o.brand
                                     AND r.no_of_stores > '4' AND o.price > '100.0'
                           GROUP BY  p.name, r.brand")
```

```
val query4DF = productDF.join(offersDF,offersDF("id") === productDF("id")).join(retailerDF,retailerDF("brand") === offersDF("brand")).filter("no_of_stores > '4'").filter("price > '100.0' ").groupBy(productDF("name"),retailerDF("brand")).agg(avg(offersDF("price")))
```

Overall: 20 points

Good luck!