Exercise Sheet 2, 2020

6.0 VU Advanced Database Systems

April 24, 2020

Exercise 1 (Costs of MapReduce) *[1 points]*

a) The purpose of a combiner is to do some reduce functionality already after the mapping phase. Since we do not use them in this case, all reduce functionality needs to be implemented in the reducers.

Each reduce task receives a different (key,value list) pair. So there is likely to be a skewed processing time in the reduce tasks, as the keys (i.e. words) will certainly differ in the length of their value lists.

What we do not know, though, is how the reduce tasks are distributed to the specific reducers (we use "some number" of reducers). The lower the number of reducers used, the less will be the skew with respect to the reducers.

b) The skew is unlikely to be significant, since the number of keys (i.e. words) will be significantly larger than the number of reduce tasks. On the other hand, when 10,000 reduce tasks are used, there will probably emerge a significant skew. Again, similar to a), if we are concerned with the skew of the reducers, this will depend on how many reducers we use.

c) Adding a combiner after the mapper will not affect the skew, if we assume that the relief provided by the combiner is the same (or nearly the same) for all reduce tasks. If that is not the case, i.e. if the combiner provides relief proportional to the lengths of the value lists, then the skew will be severly reduced.

d) Per se, the communication cost will not increase by increasing the number of reduce tasks. However, it might be sensible to assume that we also increase the number of reducers proportionally to the number of reduce tasks. In this case, the communication cost will certainly increase.

Since the replication rate depends on the mappers but not the reducers, it would not change in this scenario. Also, since the reducer size is the maximum value list length of individual keys, it does not change in this scenario.

Exercise 2 (Relational Operations) *[2 points]*

a)
- Map function:
  - output $(r$;"R") $\forall r \in R$ and $(s$;"S") $\forall s \in S$.
- Reduce function:
  - Input is $(t;[x_1 \dots x_n])$ where x_i is either "R" or "S".
  - Counts the occurences of "R" $|x_R|$ and the occurences of "S" $|x_S|$.
  - Output (t,t) $\max(|x_R| - |x_S|, 0)$ times.
- Communication cost: since the replication rate is 1 and since there is no combiner, the communication cost is $|R| + |S|$.

b)
- Map function:
  - output (key=$(r.b, r.c)$;value=("R", $r.a$)) $\forall r \in R$ and (key=$(s.b, s.c)$;value=("S", null)) $\forall s \in S$.
- Reduce function:
  - Input is (key=$(t.b,t.c)$;values=$[(x_1, y_1) \dots (x_n, y_n)]$) where x_i is either "R" or "S" and y_i is $t.a$ if x_i ="R" and null otherwise.
  - If $\nexists x_i$ for which x_i ="S" then output (t, t).
- Communication cost: since the replication rate is 1 and since there is no combiner, the communication cost is $|R| + |S|$.

Exercise 3 (MapReduce) *[4 points]*

a) For this job, I have a simple mapper which emits `(user_id, brand)` tuples for each input line, provided that the `event_type == "purchase"`. As sanity check, I also check that `brand` and `product_id` are non-empty.

The reducer gets a `(user_id, [brand])` tuple and emits `(user_id, fav_brand)`, where `fav_brand` is the brand with the highest occurrence in `[brand]`. In cases where multiple brands occur with the same frequency, only the first brand is emitted.

The `mrjob` package does not produce CSV-formatted output by default. However, the output format can be changed by setting `OUTPUT_PROTOCOL` in the job class. I used the `CsvProtocol` class of the `mr3px` package.

b) I opted to follow the advice in the assignment sheet and extended the job of a) for this task. The job consists of two steps, where the first is similar to a).

The mapper of step one now emits `(user_id, (month, brand))` subject to the same conditions as in a). `month` is calculated using `event_time`. Obviously, if this fails for some reason (such as an empty value), nothing is emitted.

The reducer of step one receives `(user_id, [(month, brand)])` and checks whether both the month 10 and 11 occur in the tuple list. If this is the case, the most occurring brand is emitted as in a). However, as opposed to a), the most occurring

brand is only emitted provided that the occurrences are strictly greater than the occurences of the second-most occurring brand.

The second step uses an inversion mapper and the reducer then simply emits the length of the `user_id` value list of each `brand` key.

In order to operate on both `2019-Oct.csv` and `2019-Nov.csv`, I simply call the Python script with both file paths.

c)

Job	Map input records	Map output records	Replication rate	Reducer output records
a)	42448765	684635	1.61%	326222
b) step 1	109950745	684635	0.62%	40946
b) Step 2	40946	40946	100%	866

Exercise 4 (Hive Exercise) *[4 points]*

a) The tables were set up as managed tables and not as external tables. I generally attempted to find fitting types and used `bigint` for all ids. For the `ROW TYPE`, I initially wanted to use the OpenCsvSerde. However, I encountered problems with loading the users and posts CSV files using this. Also, it resulted in the tables being forced to use the `string` type for each column, which I aimed to avoid. Thus, I simply used `DELIMITED` with a few clauses. I also took care to skip the header.

The only other minor problem were timestamps in the `postlinks` table. I had to explicitly define the timestamp format in order for the loading to work.

b) The query took 2m1s to complete using the normal tables. I first tried to improve this by using partitions. The only viable candidate in the query for partitions is `users.reputation`, as this can be easily split on the condition `u.reputation > 100`.

However, it is not possible to simple define a condition as partition value:

```
insert into table users_partitioned partition (reputation > 100)
select * from users where reputation > 100;
```

Therefore, I used 100 and 101 as partition "markers" and put the tuples for which the condition holds in the 101 partition and the rest in the 100 partition. I also discard unnessecary columns in my code, which makes it easier to read, but obviously does not affect the query. With this optimization, the query runtime is reduced to 1m25s.

Regarding buckets, it quickly became appearant that using them on primary keys does not make any sense, since the key is unique for every tuple. But it could make sense for foreign keys, as they can be the same for multiple tuples.

As can be seen in Table **??**, `posts.posttypeid` and `posts.owneruserid` are plausible candidates for bucketing. Since there is obviously no "multi-level" bucketing in Hive, I tried both columns separately. In theory, one could bucket using both

Field	distinct values
`posts.posttypeid`	280
`posts.owneruserid`	524
`comments.postid`	175027
`postlinks.relatedpostid`	23388

Table 1: Distinct values of all columns (except reputation) occuring in the SQL query

colums, but this would only result in the buckets being created using a hash over the concatenated column values, i.e. not really useful for this query.

When attempting this, I faced the problem that `posts.owneruserid` and `posts.posttypeid` are not actually stored as `bigint`, but have some wrong string values in them. Thus, the bucketing did not really work out.

Looking at the query plan, though, shows that the clustering did not change the query significantly. This is the case, even if `set hive.optimize.bucketmapjoin = true;` is used. It could be the case that I used the wrong datatypes for clustering. I found out when consulting the documentation[1] that `bigint` works differently than `int` for this sort of task.

Finally, I tried to combine optimization for `posts.posttypeid` and `posts.owneruserid` by partitioning using the one and clustering using the other. This turned out to be a bit tricky. I first tried to partition using `posts.owneruserid`, but for some reason this exceeded the partition limit of 1000. Then, I tried partitioning using `posts.posttypeid`. This did not work out initially, as somehow `posts.body` values leaked into the `posts.posttypeid` column, which made the paths exceed the fs limit. Thus, I use only data where `posts.posttypeid` is a bigint. Even then, the

In the end, the only viable change is partitioning the `users` table. The relevant section in the original query plan is shown below:

```
TableScan
alias: u
filterExpr: (id is not null and (reputation > 100)) (type: boolean)
Statistics: Num rows: 3784748 Data size: 45416976 Basic stats: COMPLETE Column stats:
Filter Operator
predicate: (id is not null and (reputation > 100)) (type: boolean)
Statistics: Num rows: 1261582 Data size: 15138984 Basic stats: COMPLETE Column stats:
Reduce Output Operator
key expressions: id (type: bigint)
sort order: +
Map-reduce partition columns: id (type: bigint)
Statistics: Num rows: 1261582 Data size: 15138984 Basic stats: COMPLETE Column stats:
```

[1] `https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL+BucketedTables`

With partitioning, this changes into a HashTable Sink Operator:

```
TableScan
alias: u
filterExpr: (id is not null and (reputation > 100)) (type: boolean)
Statistics: Num rows: 37075 Data size: 5011264 Basic stats: COMPLETE Column stats: NO
Filter Operator
predicate: id is not null (type: boolean)
Statistics: Num rows: 37075 Data size: 5011264 Basic stats: COMPLETE Column stats: NO
HashTable Sink Operator
keys:
0 _col14 (type: bigint)
1 id (type: bigint)
```

c) The original query shown in Listing 1 contains two subqueries that render it unable to be executed using Hive:

   a) A subquery that checks whether a users upvotes are maximal among all users created after a given post

   b) A subquery which checks whether another post links to a given post as a related post

Listing 1: Original Hive query

```
SELECT p.id FROM posts p, comments c, users u, badges b
WHERE c.postid=p.id AND u.id=p.owneruserid
AND u.id = b.userid
AND u.upvotes IN (SELECT MAX(upvotes)
FROM users u WHERE u.creationdate > p.creationdate)
AND EXISTS (SELECT 1 FROM postlinks l WHERE l.relatedpostid > p.id)
AND (b.name LIKE 'Research␣Assistant');
```

I eliminated the second subquery by simply including the `postlinks` table using a join with the condition given in the subquery. Since this only returns tuples that fulfill the condition, I ensure that the condition holds.

The second subquery could be eliminated by joining the `users` table a second time and finally grouping by `users.uservotes`. Since `GROUP BY` supports comparing group keys to aggregate functions of other fields using the `HAVING` clause, one can compare `users.uservotes` to `MAX(users2.uservotes)`.

The resulting query can be seen in Listing 2.

Listing 2: Hive querydraw without subqueries

```
SELECT p.id FROM posts p, comments c, users u, users u2, badges b, postlinks l
WHERE c.postid=p.id AND u.id=p.owneruserid
AND u.id = b.userid
AND l.relatedpostid > p.id
AND u2.creationdate > p.creationdate
```

```
AND b.name LIKE 'Research_Assistant'
group by p.id, u.id, u.upvotes HAVING u.upvotes = MAX(u2.upvotes);
```

In the end, I could confirm that the subquery ist executable by Hive and I am also convinced that the originally desired result is returned. However, the performance of the query is abysmal. The runtime is on the order of hours. Thus, I gather there must be some more optimal way to translate this query. I tried using other joins like left (semi) join, but unfortunately, I was not able to come up with a better solution.

Exercise 5 (Spark in Scala) *[4 points]*

a) I aimed to make use of Spark functions as much as possible. This was relatively straightforward for the first and the last task. Only a single `collect` action has been used for those.

The median and standard deviation case was a bit more tricky. For the median, I had to use two actions. First, I needed to use `count` in order to ascertain whether there is an even or an odd number of elements in the array. Then, I had to use `collect` in order to retrieve the correct median value from the array.

For the standard deviation, I had to resort to three actions. I needed both a `count` and a `reduce` action to calculate the average. Then, I used a `reduce` action to get a sum of squared differences.

b) Query 1 The execution plan and the runtime is exactly the same.

Query 2 The conversion is straightforward. Interestingly, although the query plan is exactly the same, the SQL version consistently a bit faster. Using the Spark Web UI (SQL tab) I could find out that both HashAggregate steps are faster for the SQL version. However, this could also be due to an unstable estimate.

Query 3 I had to implement the subquery using a pseudo left semi join, as correlated subqueries are not (yet) supported using the DataFrame API[2]. The execution plans differ slightly. The most important difference is that the SQL version uses a left outer join, while the DataFrame API version uses an inner join. In particular, the SQL version has an additional Exchange step after the second HashAggregate step of the oct2 table. In practice, this difference is negligible, as both versions have similar runtimes on average.

Query 4 Surprisingly, the execution plans are exactly the same, as are the runtimes. The runtime of the query is significantly higher than the others. I also experienced higher variance in the runtime estimation.

[2] `https://issues.apache.org/jira/browse/SPARK-23945`

c) I use here both the physical query plans and the Stage tab of the Spark Web UI in order to view the partitioners. Based on that, I detect which stage has wide dependencies.

Query 1 What initially surprised me is that the count has wide dependencies. This seems to be due to the implementation of `count()`, which seems to use `groupBy`[3]. If I interpret the query execution plan correctly, one sees that Spark computes the partial counts first and then for some reason shuffles the data before adding them up. All other nodes have narrow dependencies.

Query 2 The situation is similar to Query 1, although it makes sense in this case as it performs a `GROUP BY` with hash partitioning.

Query 3 The join is implemented as sort merge join in both cases. The join - which resides in the last stage - has wide dependencies, since it uses ZippedPartitionsRDD and the Sort step depends on an Exchange step with hash partitioning. Also, both `GROUP BY` with `COUNT` operations (in stage 2 and 5 resp.) use the ShuffledRowRDD and therefore have wide dependencies. Interestingly, for the DataFrame API execution plan, the Sort step (for the sort merge join) does not have wide dependencies. All other operations have narrow ones.

Query 4 Obviously the Sort step implementing the `ORDER BY` operation in stage 7 has wide dependencies. Also, the join between the oct and nov tables is implemented as sort merge join, thus both tables have to be sorted in stage 2 and 4 resp. which have wide dependencies. And, as before, the `GROUP BY` with `AVG` operation (in stage 6 resp.) has wide dependencies. All other operations have narrow dependencies.

[3] `https://stackoverflow.com/a/47205454`

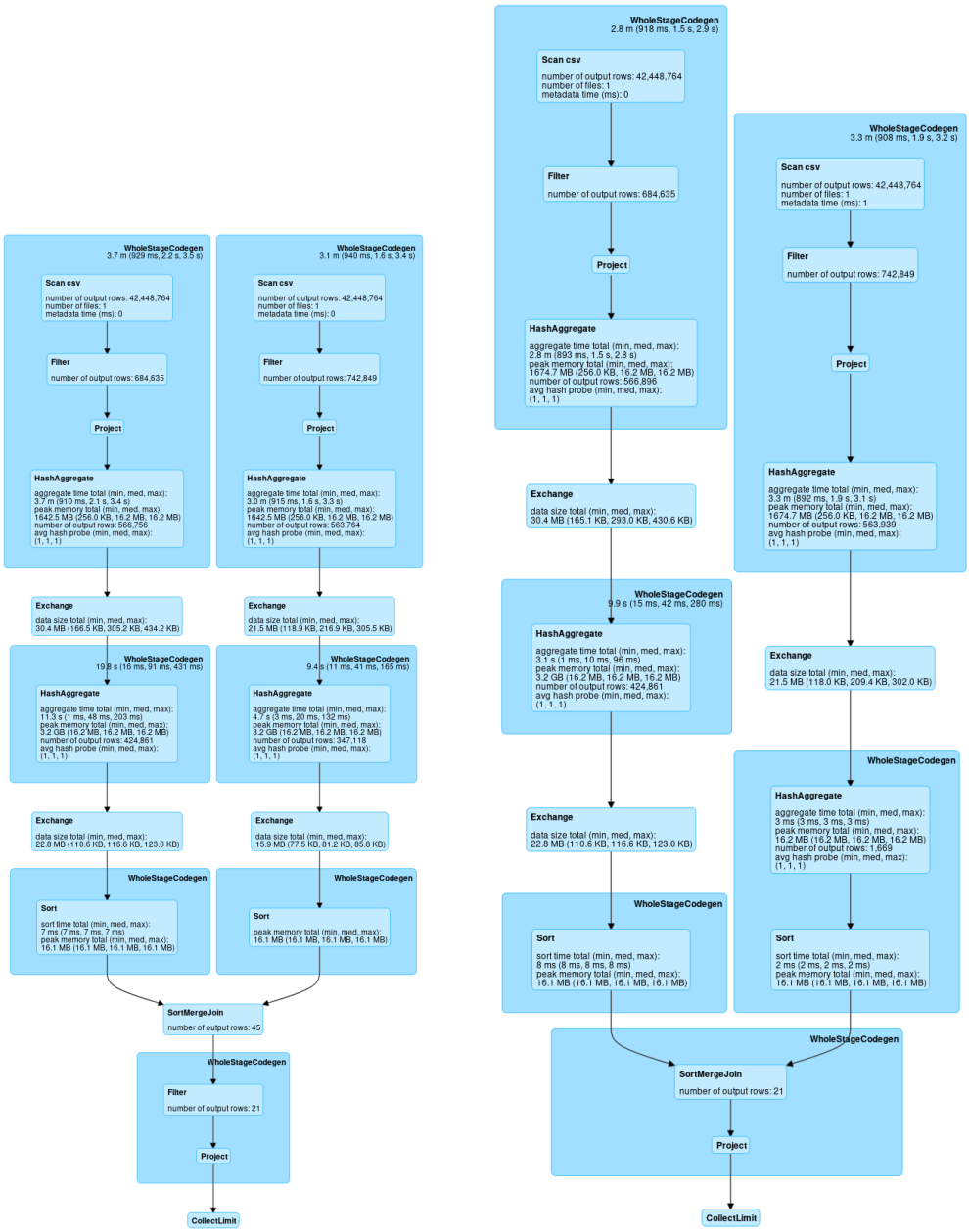

Figure 1: Execution plans for Query 3

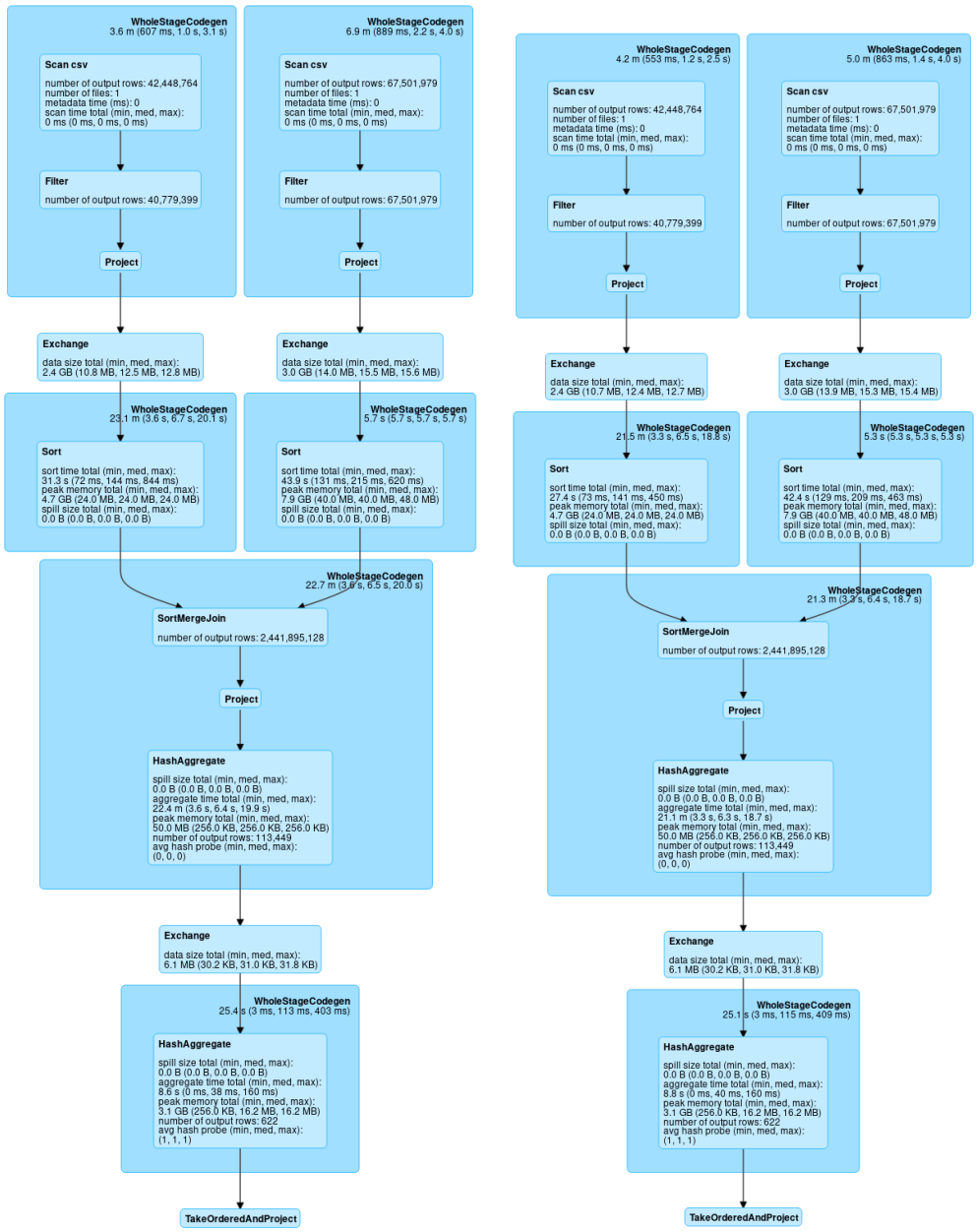

Figure 2: Execution plans for Query 4