Exercise Sheet 2, 2019

6.0 VU Advanced Database Systems

Gent Rexha (11832486), Princ Mullatahiri (11846033)

08.05.2019

Exercise 1 (MapReduce) [4 Punkte]

a) First of all, we take the input and map the author to the row in the `checkouts-by-title.csv` file. The output from the mapper is `<Author,RowContent>` for the read line. After that most of the heavy lifting takes place in the reducer, where a hashmap of all the titles of the current creator is created and for each value in our values list the number of checkouts is updated for the corresponding title. After having iterated all the values, the title with the maximum amount of checkouts is written out with it's corresponding author. Reducer output is `<Author, Most Checkouts Book Title>`.

Statistics:

- replication rate = 1
- input size = 6789 Mbyte
- output = 27 Mbyte

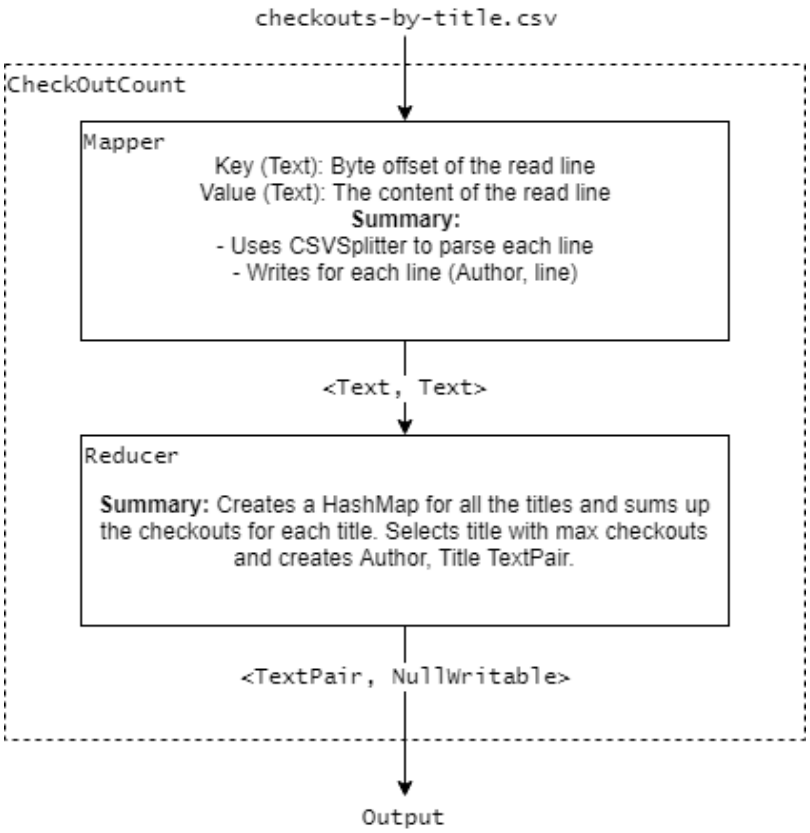

Figure 1: Diagram of the Map Reduce job to solve Task A

b) For this task we used the MultipleInputs class. It allows you to create MapReduce jobs with multiple mapper functions, each for a different input file. The results of the mappers is then sent to the same reducer

function. For the two datasets we've created two separate mappers which send the row with the author as key to the reducer `<Author, RowContent>`. The reducer then creates two HashMaps, for both checkouts and inventory. Checkout HashMap includes all the titles and sums up the checkouts for each title. After that it selects the title with max checkouts and creates Author, Title TextPair and adds PublicationYear, Subjects, ItemLocation from the second HashMap where all the locations for all titles of said Author are stored. The reducer output looks like this `<Author, Most Checkouts Book Title, Publication Year, Subjects, Item Location>`.

Statistics:

- replication rate = 1
- input = 13944 Mbyte
- output = 36 Mbyte

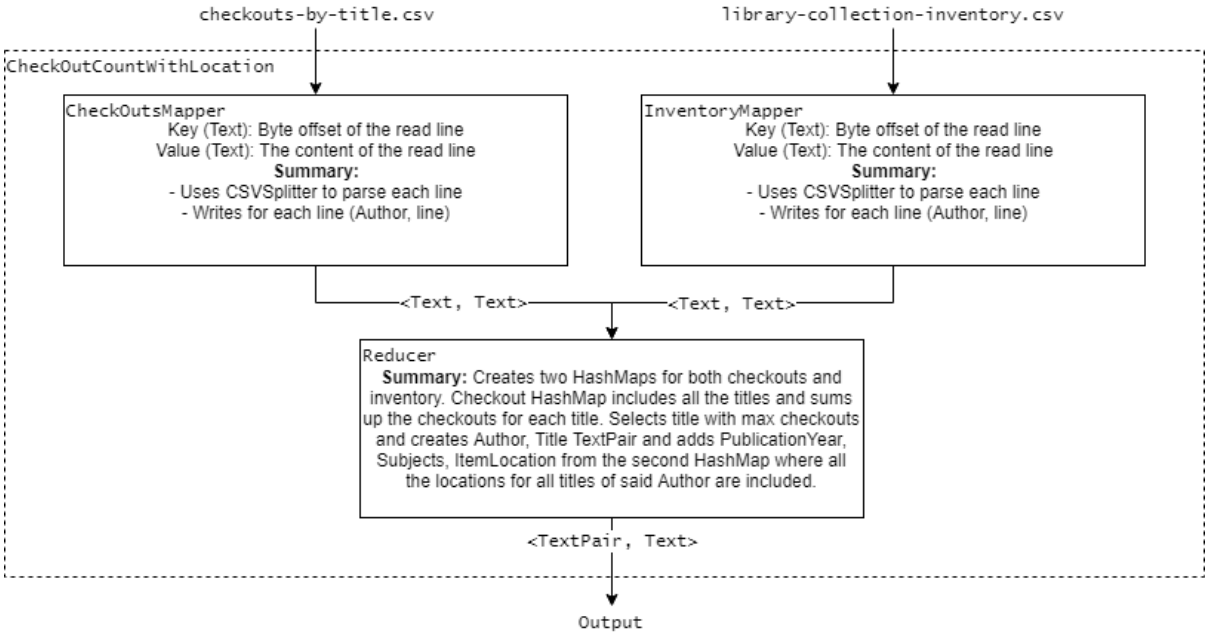

Figure 2: Diagram of the Map Reduce job to solve Task B

Exercise 2 (Costs of MapReduce) [1 Punkte]

a) In the Reducer, if not using a combiner, there is some skew to be expected, given that different keys can receive different amounts of values, resulting in different processing times. Key A can have many values, for example, but key B can have very few or none, resulting in more calculation for key A values and therefore more processing time. This is till dependent on the `key,value` data distribution, because in the extreme case where all of your values are of the same key, the number of reducers doesn't matter. In this case a custom partitioner might make more sense.

b) Considering a small number of reducers, more randomized key values are expected at different reducers and a more even distribution of one key's intensive processing values. This also leads to less parellesim and may take longer. If the number of reducers is increased, the likelihood that only one reducer will end up with a pair where many values are also higher. Therefore, the skewedness is increased to a very important level. On the other hand, a maximum of parallelism is achieved, but the problem of overhead computing comes with this.

c) If we use a combiner in the WordCount MapReduce program then we'd have less reducer calculation and more for mapping phase, basically we'd calculate sum of word occurrences for each mapper node in a document. And we don't expect to be a significant skew because reducer would have less work to do. The best way to avoid skewing would be to use combiner and partition where we determine which task reduction receives which key and its values?list, to partition the key we could use a hash function and this would ensure that all pairs with the same key go to the same reducer.

d) Communication costs are made by the size of the input file, the sum of the sizes of all files passed from the map process to reduce processes and the sum of the output sizes of reducer processes, so basically it does not depend on the number of reducers we have, because even if we have 1 reducer or if we have 100000 reducers, we would still have the same size of the input file. We'd still have the same file size input, the same sum of all file sizes passed from map to reduce, and the sum of all reducer outputs.

Exercise 3 (Relational Operations) [2 Punkte]

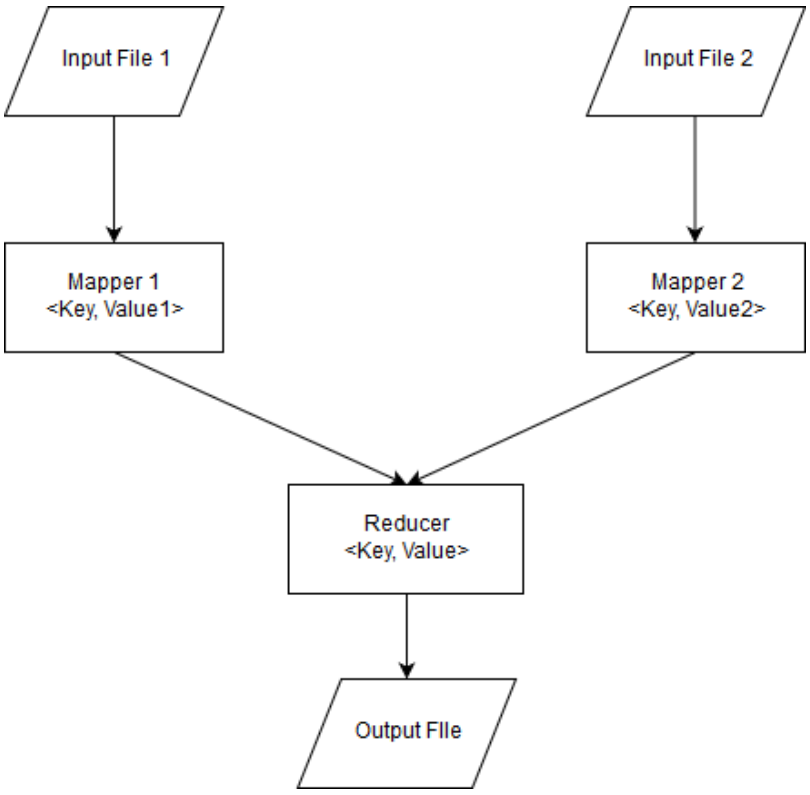

Figure 3: Mapreduce Sketch for Bag Union and Difference

a) First we get all values from the same key in reducer, then we iterate through each value and increment sum variable, then we simply write to output key and sum as value. Bag Union Pseudo-code:

Map

The map function maps each `value` to `(value, Null)`.

Reduce

```
sum = 0

foreach val:value
    sum += 1

context.write(key, sum)
```

b) Bag Difference Pseudo-code:

Map:

```
// For tuples in R:
value -> (value, 1)

// For tuples in S:
value -> (value, -1)
```

Reduce:

```
sum = 0

foreach val:values
  sum += val

if sum > 0
  context.write(key, sum)
```

Communication cost = input file size + 2 * (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.

Exercise 4 (Hive Exercise) [4 Punkte]

a) For task A we have to create tables and populate them with data given in CSV files. So first we create the table, we tell that values are seperated by comma and then give the location of CSV file.

```
CREATE EXTERNAL TABLE IF NOT EXISTS group11.badges (
id INT,class STRING, date DATE, name STRING, tagbased STRING, userid INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/home/adbs/2019S/shared/hive/badges.csv';
```

b) Since hive does not allow new, slightly renamed versions to partition (or bucket) existing tables have been created. Creation is a multi-step process: the first step is to create a new, empty table. Using an `INSERT OVERWRITE` statement, this table is then populated. Here the selection order is important and must match the order in which the columns of the table have been declared. The columns of partitioning come last.

```
CREATE TABLE badges_p (id INT, class INT, timestamp STRING, name STRING, tagbased BOOLEAN)
PARTITIONED BY (userid INT);
INSERT OVERWRITE TABLE badges_p PARTITION (userid) SELECT id, class, timestamp,
  name, tagbased, userid FROM badges;
```

```
CREATE TABLE users_p (id INT, aboutme STRING, accountid INT, creationdate STRING,
displayname STRING, downvotes INT, lastaccessdate STRING, location STRING,
profileimageurl STRING, upvotes INT, views INT, websiteurl STRING)
PARTITIONED BY (reputation INT);
INSERT OVERWRITE TABLE users_p PARTITION (reputation)
SELECT id, aboutme, accountid, creationdate, displayname, downvotes, lastaccessdate,
location, profileimageurl, upvotes, views, websiteurl, reputation
FROM users;
```

For columns with not too many distinct values, partitioning is useful. Instead of speeding up the query, too small splits can add overhead. This disqualifies all primary and foreign keys in essence, as they would create a vast number of splits.

This leaves only the reputation of the user, so we used to partition this column. Since only about one third of all users fulfill the attribute of `reputation > 100` this looked promising. However, in fact, our query time increased from 67.05 seconds to 74.47.

Dividing tables into buckets works like partitioning. First the table is created, then another query is filled in. Sample code for the `users` and `comments` tables:

```
CREATE TABLE users_b (id INT, aboutme STRING, accountid INT, creationdate STRING,
displayname STRING, downvotes INT, lastaccessdate STRING, location STRING,
profileimageurl STRING, upvotes INT, views INT, websiteurl STRING, reputation INT)
CLUSTERED BY (id) INTO 32 BUCKETS;
INSERT OVERWRITE TABLE users_b
SELECT id, aboutme, accountid, creationdate, displayname, downvotes, lastaccessdate,
location, profileimageurl, upvotes, views, websiteurl, reputation
FROM users;
```

```
CREATE TABLE comments_b (id INT, creationdate STRING, postid INT, score INT,
text STRING, userdisplayname STRING, userid INT)
CLUSTERED BY (id) INTO 32 BUCKETS;
INSERT OVERWRITE TABLE comments_b
SELECT id, creationdate, postid, score, text, userdisplayname, userid
FROM comments;
```

Applying buckets to the table of `comments` reduced the time of the query by about 5 seconds. Doing the same for `users` again didn't yield any significant results.

Next different bucket sizes were tested.

- No buckets: 96s
- 8 Buckets: 119s
- 32 Buckets: 99s
- 64 Buckets: 97s
- 128 Buckets: 91s

c) Because Hive does not support subqueries and also does not support unequal semi join we have to replace `EXISTS` with `JOIN`. And for count we place it in a table and filtering is applied in the WHERE clause.

```
SELECT p.id
FROM comments c, badges_tmp b, posts p, postlinks pl, users u,
(SELECT COUNT(upvotes)as nrupvotes FROM users) k
WHERE c.postid=p.id
AND u.upvotes + 3 >= k.nrupvotes
AND u.id=p.owneruserid
AND pl.relatedpostid > p.id
AND u.creationdate = c.creationdate
AND u.id = b.userid
AND (b.name LIKE 'Autobiographer');
```

Exercise 5 (Spark in Scala) [4 Punkte]

a) All solutions are inside the `.ipynb`.

b) As displayed in Figure 9, we can see that most of the time the Dataframe API variant of the query outperforms the SQL version in duration. Although analyzing and interpreting the results didn't make much sense. Most of the times the explanation plans look, and are, the same.

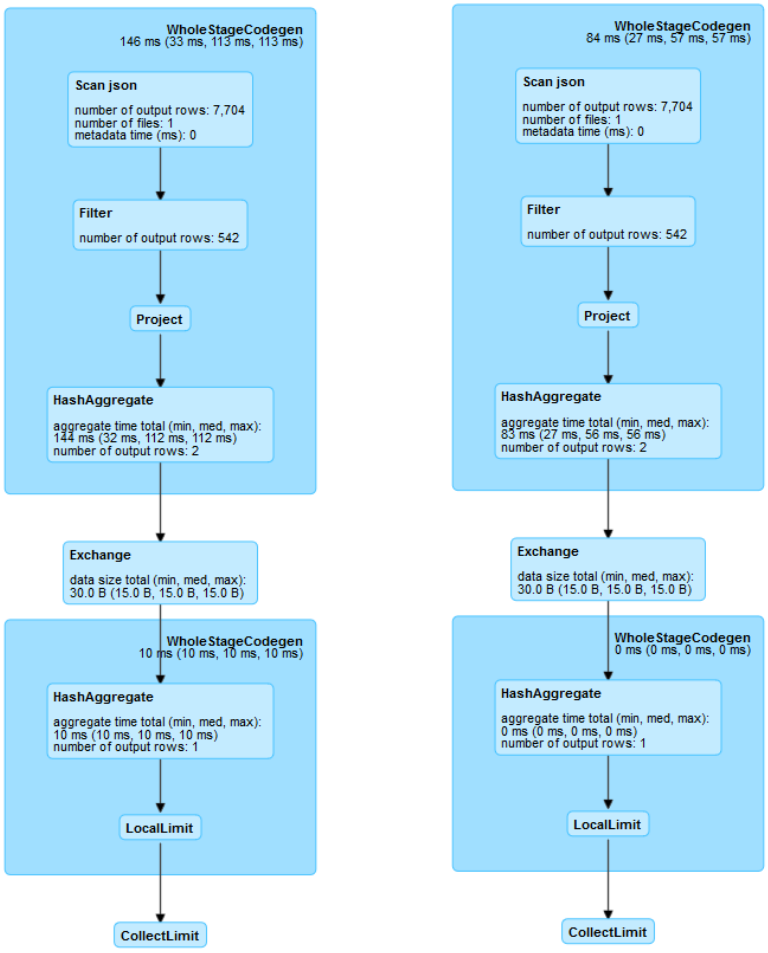

Figure 4: LEFT: SQL DAG Diagram. RIGHT: Dataframe API DAG Diagram

c) Narrow dependencies: Each partition of the parent RDD is used by at most one partition of the child RDD.

```
[parent RDD partition] ---> [child RDD partition]
```

Wide dependencies: Each partition of the parent RDD may be used by multiple child partitions.

```
                        ---> [child RDD partition 1]
[parent RDD partition] ---> [child RDD partition 2]
                        ---> [child RDD partition 3]
```

Concluding from the DAG diagrams, all of the queries have narrow dependencies.

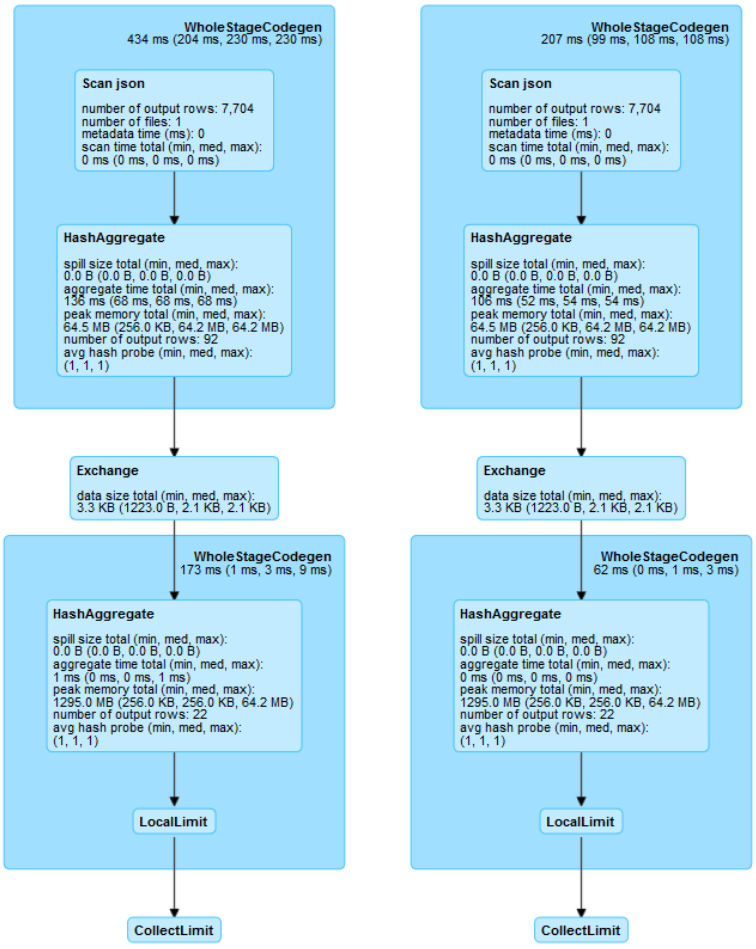

Figure 5: LEFT: SQL DAG Diagram. RIGHT: Dataframe API DAG Diagram

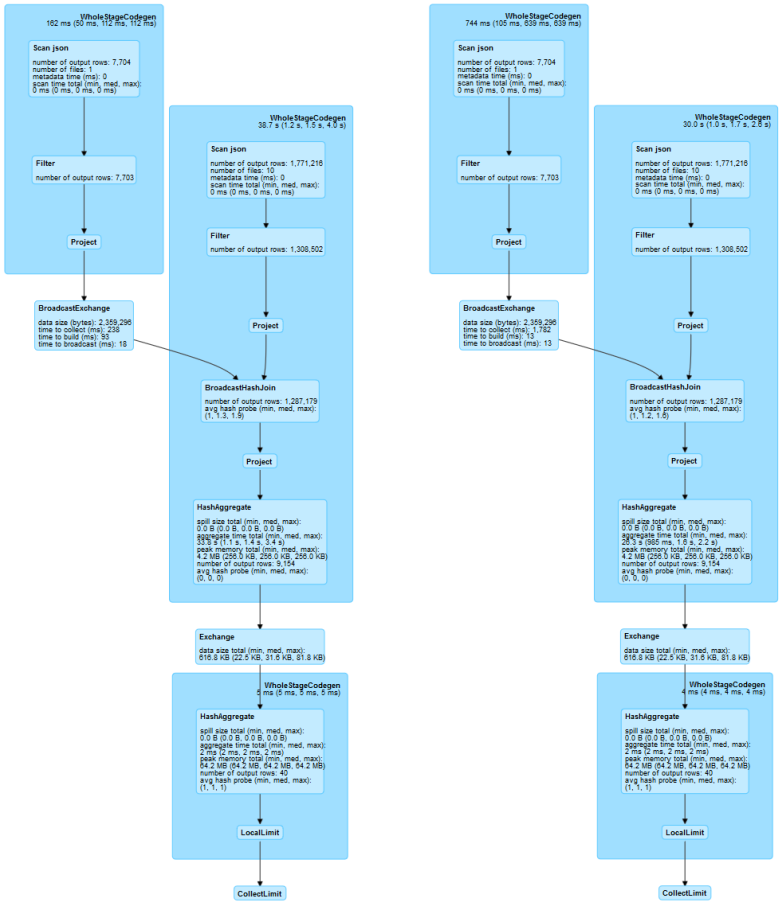

Figure 6: LEFT: SQL DAG Diagram. RIGHT: Dataframe API DAG Diagram

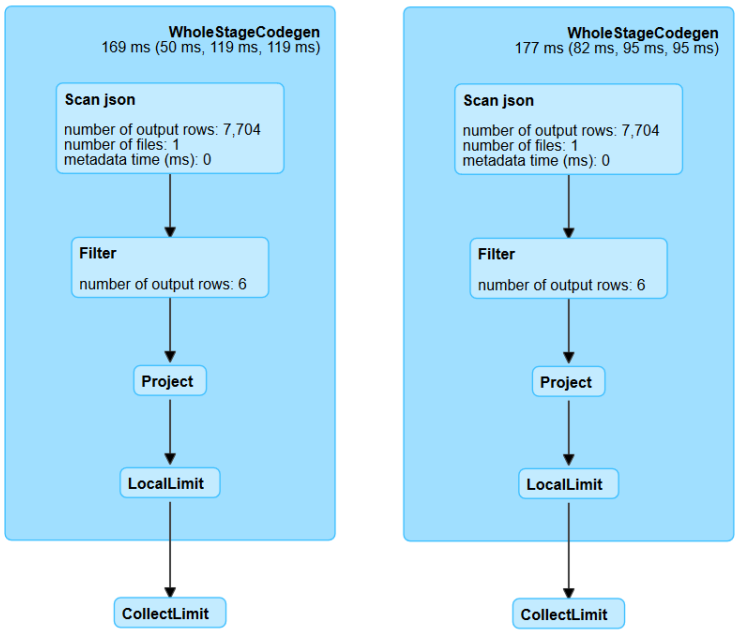

Figure 7: LEFT: SQL DAG Diagram. RIGHT: Dataframe API DAG Diagram

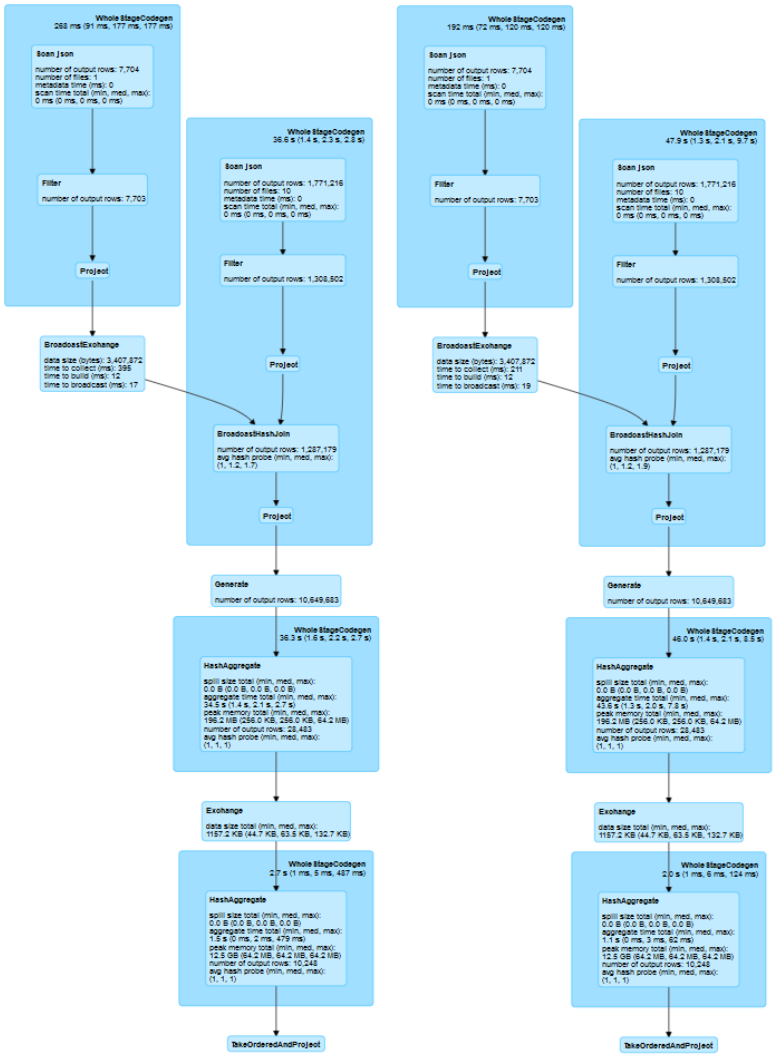

Figure 8: LEFT: SQL DAG Diagram. RIGHT: Dataframe API DAG Diagram

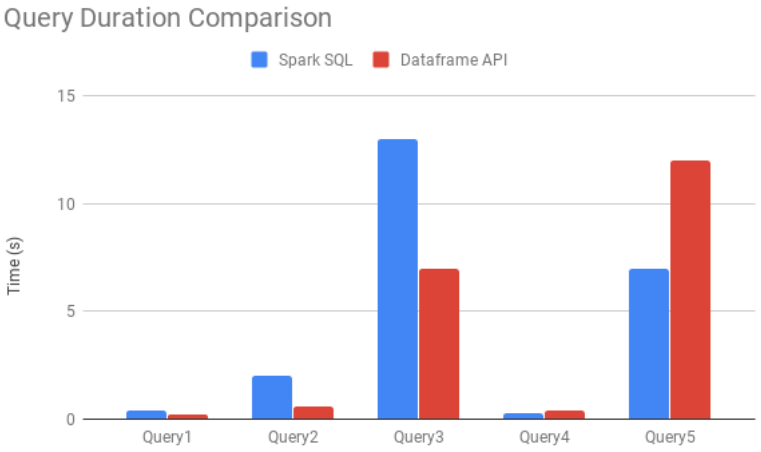

Figure 9: Query Duration Comparison

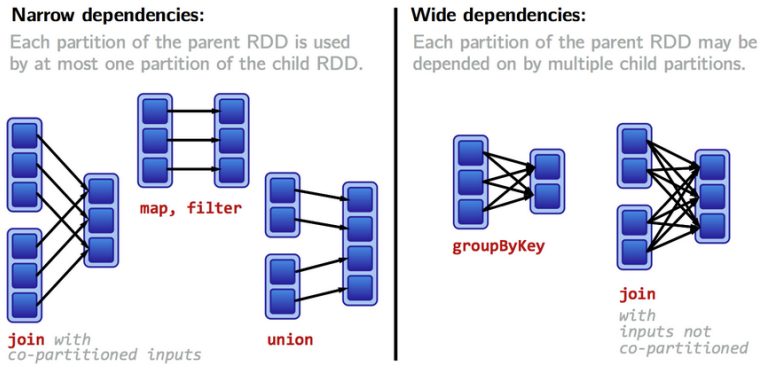

Figure 10: Wide and Narrow Dependencies Visualisation