DATA3404 Documentation

SID: 490196302 480142841 490023824

Job Design Documentation

Task 1 (dataframe API)

Firstly, import datasets with a schema using the "**read.csv()**" function. And then all the aircrafts which are made by the "Cessna" manufacturer should be selected by "**filter()**". "**filter()**" can select the tuples which satisfy the condition we set. Next the "flights" dataset and "aircrafts" dataset are joined together based on the key "tail_number" using "**join()**". "**join()**" can let one dataframe Join with another DataFrame and Inner join is used here.

```
#join counts and CESSNA_ma
join_DF = CESSNA_ma.join(DF_Flights_clean, on=['tailnum'], how='inner')
```

Then, the manufacturer's name needs to be changed to the required format using the "**initcap()**" function. "**initcap()**" can convert the first letter of each word to uppercase.

```
join_DF = join_DF.withColumn('manufacturer',f.initcap(f.col('manufacturer')))
```

Finally the top 3 are selected using "**limit()**". "**limit()**" can return a DataFrame with top N rows which N can be set by users.

Task 1 (RDD API)

Firstly, import datasets with a schema using the "**read.csv()**" function. Next two RDDs are created by the "**.rdd**" method. Then all the aircrafts which are made by the "Cessna" manufacturer are selected by "**filter()**". And then "tail_number" are set as the keys of two RDDs by the "**map()**" function. "**map()**" can mutate each tuple with the method we assign. Then the "**aggregateByKey()**" function is used for implementing "Cessna XYZ" form.

```
def seq(a,b):
  if str(b)[26] == 'T':
    return str(a) + ' ' + str(b)[27:30]
  return str(a) + ' ' + str(b)[26:29]

def combine(a,b):
  return a
key_manufact_filter = key_manufact_filter.aggregateByKey('Cessna', seq, combine)
```

"**aggregateByKey()**" can aggregate the values of each key, using seq & combine functions and a neutral "zero value". seq & combine functions and "zero value" should be classified by the users. Here "zero value" is "Cessna" and "Cessna" would be added at the start of each tuple. seq functions can select the first three digit number, and combine functions just combine the outcome together. And then "flights" RDD and "aircrafts" RDD should be joined according to "tail_number" by "**join()**" function. "**join()**" can be used to join two RDDs when the key is matched. Then the number of departures for each model would be calculated by "**map()**" and "**reduceByKey()**" functions. "**map()**" would assign each Cessna model an integer 1 if it appeared in RDD. And "**reduceByKey()**" would sum up all the 1 based on each Cessna model. Which are the number of departures for each Cessna model.

```
#count number
count = newRdd.map(lambda x:(x[1][0], 1))
count = count.reduceByKey(lambda x,y: (x+y))
```

Finally use "**sortBy()**" and "**take()**" functions to select the top 3 Cessna models. "**sortBy()**" would sort the tuple based on the column we choose, and "**take()**" can select the top tuple as much as we want.

Task 2 (Dataframe API)

Firstly, write schema and import datasets with schema using “**read.csv()**”. And then all the airlines which are in the US and all the US airlines which are in the certain year are selected by “**filter()**”. And then we convert the difference between actual departure time and scheduled departure time by “**substr()**”, “**cast()**” and the basic math operator. “**substr()**” can select the first two strings, which represent hours and the last two strings which are minutes. “**cast()**” can convert string to integer to implement math operation “hours*60 + minutes”. We assume no flight departs more than a half day earlier than scheduled. Which means all the flights which differ less than -720 minutes, they are the flights which delay over mid-night, their actual departure time should be added a day. Then we filter the delayed flights using the “**filter()**” function.

```
  #convert to minutes
  DF_Flights_clean = DF_Flights_clean.withColumn('scheduled_departure_time', f.col('scheduled_departure_time').substr(0,2).cast("int")*60
+ f.col('scheduled_departure_time').substr(4,2).cast("int"))
  DF_Flights_clean = DF_Flights_clean.withColumn('actual_departure_time', f.col('actual_departure_time').substr(0,2).cast("int")*60 +
f.col('actual_departure_time').substr(4,2).cast("int"))
  #filter next_day's delay
  DF_Flights_clean = DF_Flights_clean.withColumn("actual_departure_time", f.when(f.col("actual_departure_time").cast("int") -
f.col("scheduled_departure_time").cast("int") < -720, f.col("actual_departure_time").cast("int") +
1440).otherwise(f.col("actual_departure_time").cast("int")))
```

Then we calculate the delay time by “**withColumn()**” and basic math operation. “**withColumn()**” can be used for creating a new column or updating an existing column with the way we want. And then we join the delayed flights with “carrier_code” using “**join()**”. Finally we calculate the number of delay, mean, max, min of delay time based on each airline using “**count()**”, “**min()**”, “**max()**” and “**avg()**”.

Task 2 (RDD API)

Firstly, import datasets with a schema using the “**read.csv()**” function. Next two RDDs are created by “**.rdd**”. Then all the airlines which are in the US and all the US airlines which are in the certain year are selected by “**filter()**”. And then “carrier_code” are set as the keys of two RDDs by the “**map()**” function. Then join two RDDs by the “**join()**” function based on the key “carrier_code”. Then the delay time is calculated by “**map()**” and “**int()**” functions. “**int()**” can change the data types to integer so that we can calculate the delay time using formula hour*60 + minutes. We assume there is no flight departing early over 12 hours. For the flights which are early over 12 hours, they are treated as delayed over mid-night and plus 24 hours for them using “**map()**” and a user-defined function. If the difference between actual departure time and scheduled departure time is less than -720 minutes, we plus 24 hours, otherwise do not change anything. And then mutate each tuple with the function defined above by “**map()**”. Next all the delayed flights are selected by the “**filter()**” function

```
  #calculate delay
  count = newRdd.map(lambda x:(x[1][0], (int(x[1][1][1][0:2])*60 + int(x[1][1][1][2:4])) - (int(x[1][1][0][0:2])*60 + int(x[1][1][0]
[2:4])) ))
  #filter delay next day ?? Assume
  count = count.map(lambda x: (x[0], x[1]+24*60) if x[1] < -12*60 else (x[0], x[1]))
  #filter delayed flights
  count = count.filter(lambda x: x[1] > 0)
```

To calculate the number of delayed flights, we used “**map()**” and “**reduceByKey()**”. “**reduceByKey()**” can add the values if the key of two tuples are matched. For average delay time, we use “**map()**” and “**reduceByKey()**” to calculate the total delay for each airline first and then divide by their delay number. We calculate the maximum delay and minimum delay for each airline using “**aggregateByKey()**” and “**mapValues()**”. “**mapValues()**” would put each value in a key-value pair, it won’t break any partitioning and change any key.

Task 3 (Dataframe API)

Firstly, import datasets with schema using “**read.csv()**”. And then all the airlines which are in a user-specified country are selected by “**filter()**”. And then three datasets are joined by “**join()**” using “carrier_code” and “tail_number” as keys. Then we combine “manufacturer” and “aircraft_type” using the “**concat_ws()**” function. “**concat_ws()**” can paste two strings. And then we calculate the count of each type and group by airline using “**select()**”, “**groupBy()**” and “**count()**”. “**groupBy()**” can group the tuple according to the column we assign so that “**count()**” can be implemented in each group. And then top 5 aircrafts for each airline are found which “rank” less or equal to 5 using “**filter()**”. “rank” is

created by “**rank()**” which can rank all the tuples in each group based on a column. To drop the none, “null” should be converted to “None” by “**fillna()**”. And then drop them with “**array_remove()**”. “**array_remove()**” can remove the string we assign. Finally we convert data to required format with a UDF “**array_to_string()**” .

```
def array_to_string(my_list):
  return '[' + ','.join([str(elem) for elem in my_list]) + ']'
```

This function can separate each aircraft’s name by comma and store them in a list for each airline. Finally, store result in csv using “**write.csv()**”

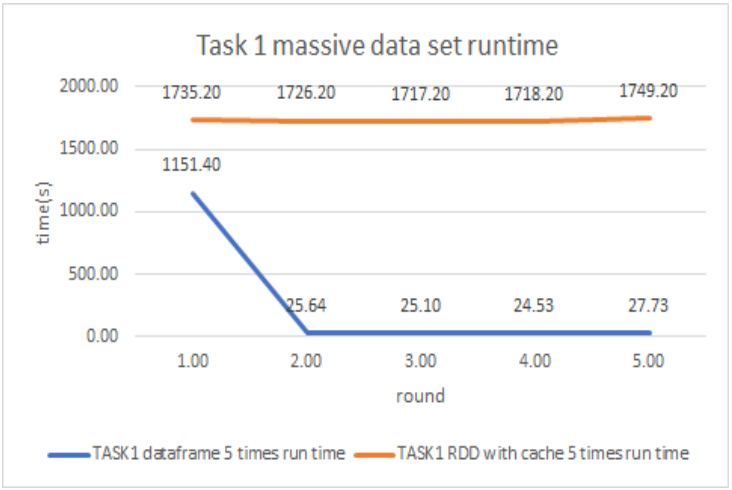
Task 3 (RDD API)

Firstly, import datasets with a schema using the “**read.csv()**” function. Next three RDDs are created by the “**.rdd**” method. . Then all the airlines which are in a user-specified country are selected by “**filter()**”. Then join airlines and flight with “**join()**” based on “carrier_code”. Next, the number of uses for each tail_number is calculated by “**map()**” and “**reduceByKey()**”. And then using the “**sortBy()**” function to sort the aircrafts for each airline based on their number of uses. And then the name of aircrafts in aircrafts RDD should be converted to the required format like “Boeing 787” by “**map()**”. Then the aircrafts RDD is joined with the key “tail_number”. And then count numbers for each aircraft type group by airlines using “**map()**” and “**reduceByKey()**”. Then we sort the tuple based on both airlines’ names and the number of uses of each aircraft type using “**sortBy()**”. And then limit top5 for each airline using “**groupByKey()**” and select from 0 to 5.

Run time analysis:

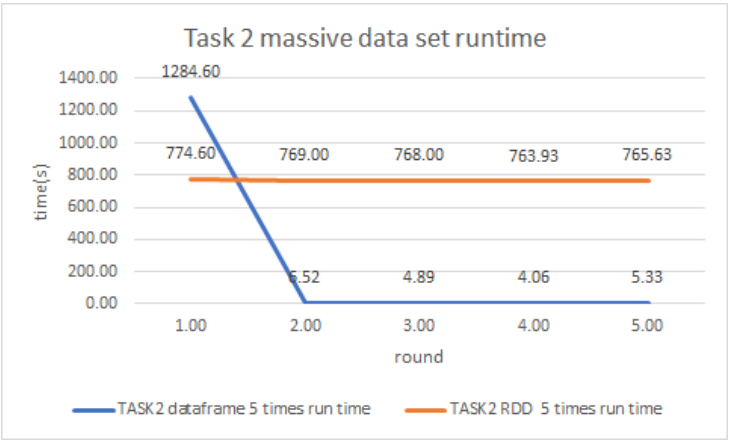
Task1 runtime table:

small	TASK1 dataframe run time	TASK1 RDD run time
1	3.68	8.78
2	1.97	6.91
3	1.59	5.90
4	3.64	6.33
5	2.84	6.83
medium		
1	22.30	20.40
2	3.20	19.97
3	2.66	22.97
4	3.90	22.25
5	2.36	23.54
large		
1	163.80	197.40
2	4.36	205.20
3	5.24	234.00
4	4.63	204.00
5	5.46	222.30
massive		
1	1151.40	1735.20
2	25.64	1726.20
3	25.10	1717.20
4	24.53	1718.20
5	27.73	1749.20



Task2 runtime table:

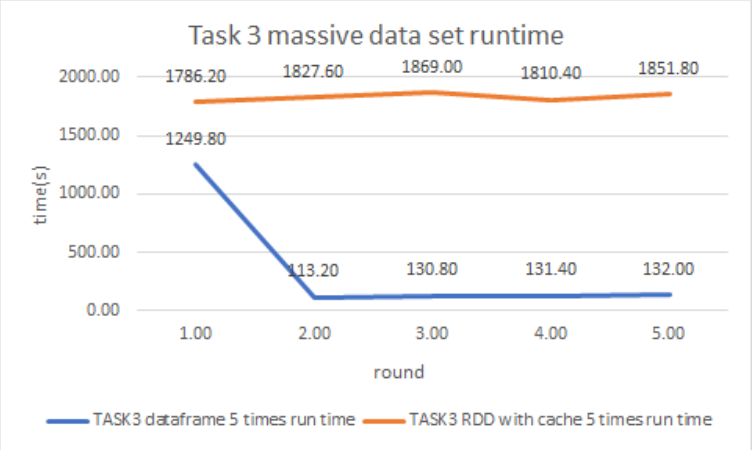
small	TASK2 dataframe run time	TASK2 RDD run time
1	10.23	7.68
2	5.00	5.02
3	5.58	4.78
4	4.47	5.03
5	5.42	5.06
medium		
1	34.40	16.30
2	2.76	16.47



3	2.69	17.35
4	2.18	16.42
5	2.18	16.45
large		
1	189.60	345.00
2	2.23	185.00
3	2.03	165.00
4	2.46	187.80
5	2.67	182.07
massive		
1	1284.60	774.60
2	6.52	769.00
3	4.89	768.00
4	4.06	763.93
5	5.33	765.63

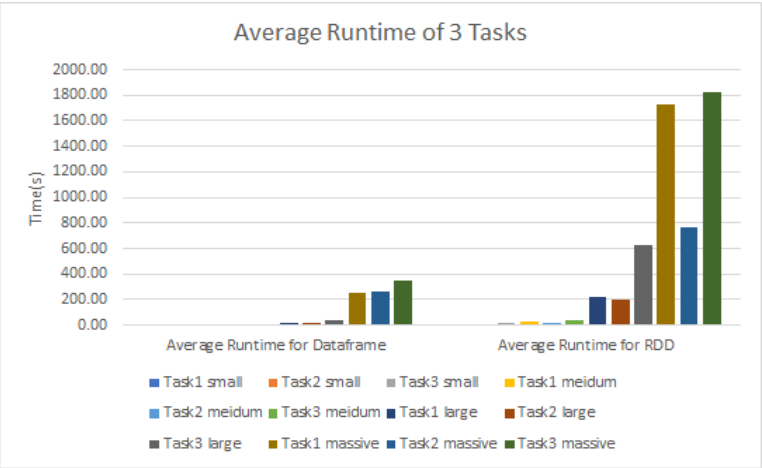
Task3 runtime table:

small	TASK3 dataframe run time	TASK3 RDD run time
1	10.25	17.44
2	6.67	15.84
3	5.51	15.64
4	6.53	15.19
5	5.53	14.67
medium		
1	38.20	35.20
2	4.14	35.11
3	2.81	36.56
4	3.77	34.44
5	2.88	34.85
large		
1	363.60	601.80
2	6.30	576.00
3	6.64	603.00
4	6.37	628.20
5	6.56	655.80
massive		
1	1249.80	1786.20
2	113.20	1827.60
3	130.80	1869.00
4	131.40	1810.40
5	132.00	1851.80



The average execution times:

	Average Runtime for Dataframe	Average Runtime for RDD
Task1 small	2.77	6.68
Task2 small	5.36	5.12
Task3 small	6.50	15.01
Task1 meidum	4.74	23.96
Task2 meidum	5.68	16.46
Task3 meidum	6.92	35.13
Task1 large	20.27	224.73
Task2 large	21.17	195.42
Task3 large	42.27	631.06
Task1 massive	250.88	1729.20
Task2 massive	261.08	768.23
Task3 massive	351.44	1829.00



The Communication costs:

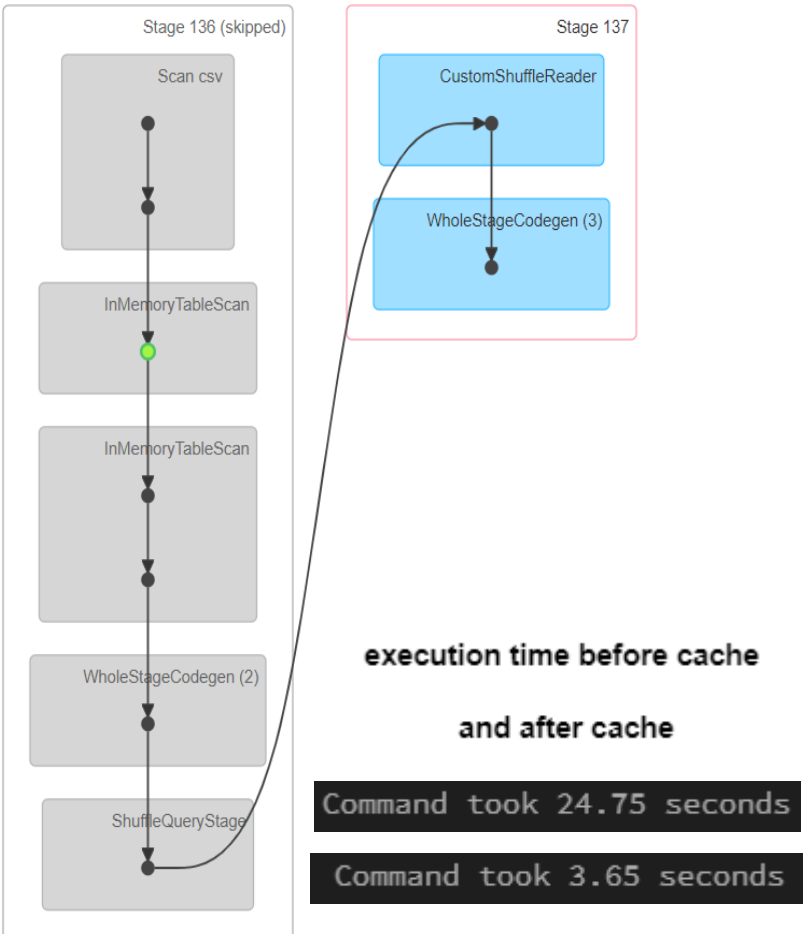
massive	intermediate result size for Dataframe	intermediate result size for RDD
TASK 1	0 KB	462 MB
TASK 2	0 KB	22 MB
TASK 3	747 KB	820 MB

As shown in above tables and diagrams, the method to compute the run time for us is to record 5 rounds of running time for each of these queries. We made few assumptions on these queries: Larger the file size, longer the time for query to execute; RDD performance will be worse than the dataframe performance and larger the intermediate results the longer the time it's gonna run. The results of running these three tasks at different sizes show that our assumptions generally hold true.

First is the size of the file will affect the runtime which easily shows in comparison that a massive file takes the longest time to run and a small size file takes least time to run.
The second assumption also holds true since RDD is based on hadoop Mapreduce that use the materialization and dataframe is based on relational DBMS that use the pipelining. The difference between these two is that Materialization will materialize the complete intermediate result for each operator we use in parallelism and Pipelining does not, this explains why we have such a massive intermediate result size for RDD method but none or relatively small size for dataframe method. The performance of these two implementations are quite different since the RDD implementation needs to send and redistribute to the target nodes when doing the join operations in all three tasks since it has a large intermediate results thus we assume it used the distributed-shuffle join method and this would create a cpu overhead and thus leads to a worse performance. On the other hand dataframe implementation uses meta-operators into query plans that hide the communication thus no cpu overhead and also no intermediate result and it can not be using the distributed-shuffle join, since we do the push down for all three tasks that we assume the join method here is a broadcast join, thus lead to a better performance.

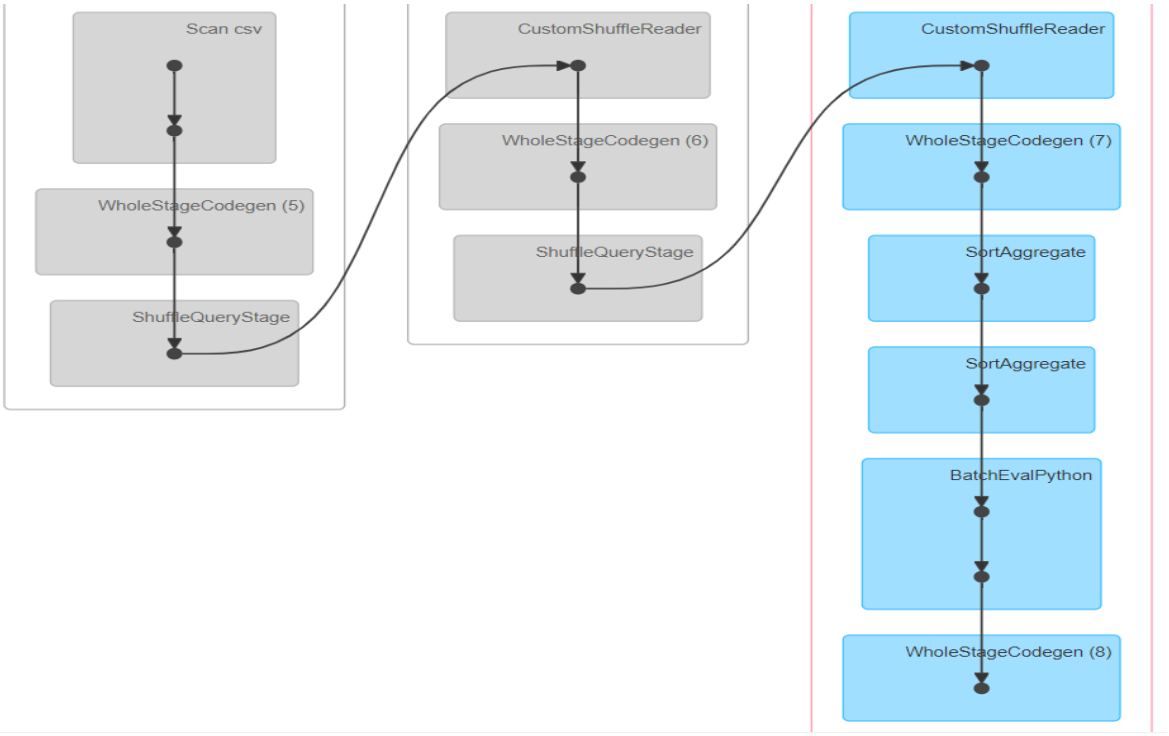
Justification of turning decisions or optimizations:

1. Cache. Cache is one of the most important changes in our execution plans for Dataframe API. Before we use cache to store datasets from the given csv files, we found that it takes a long execution time since the csv files would load for every time we ran our project codes. When those files contained a huge amount of data, it took several minutes to execute. However, after we researched the DAG visualization graph, we realized that there is no need to re-execute the loading stage by using cache. It will build a cache for datasets after the first execution time and from the second time, data can be fetched from cache and it is much faster than fetching from disk. Thus from our runtime analysis it shows that in Task 1 massive dataset the first execution time took nearly 20 minutes and from the second time it only took 5 seconds. It is also consistent in Task 2 and Task 3 for all small, medium, large and massive datasets. For example, we run Task 1 medium file before cache and after cache. we can notice that execution for “after_cache” is much faster than “before_cache”.



2. Calculating delayed flights. In Task 2 we are asked to calculate delayed flights and it is complex to analyze those flights delayed to the next day. First we are going to use a timestamp. Since it has been given flight date, scheduled departure time and actual departure time, we just need to combine these three columns and formula as the timestamp type. However, it is not clear if the flight date is the scheduled one or the actual one. Thus we decided to transform time to minute. We split the time using ":" and use the formular hours * 60 + minutes to calculate the minutes. Then we compare the minutes of scheduled time and actual time. If actual time is greater than scheduled time or actual time + 12*60 is smaller than scheduled time, we can consider it as delayed because there exist flights that are delayed to the next day and almost no flights will take over 12 hours. Therefore based on this consideration we can easily find delayed flights.

3. Using optimal join strategies. In Task 3 we needed to join three datasets and we came up with several different join strategies to reduce the time cost. We did equality broadcast joins with 1 column twice for these three csv files and in order to reduce I/O, we use the smaller dataset to join the bigger dataset and use a left-deep join plan. Since it allows us to generate all fully pipelined plans, it is much cheaper for I/O in memory than materialization and saves time.



4. We are going to choose the Dataframe API instead of RDD API for our project. Compared with Dataframe API, RDD API does not have superiority in execution efficiency and Dataframe API uses much less time cost based on our running time analysis. Using RDD may cause many shuffle query stages. While doing MapReduce, shuffle is necessary because frequent disk I/O operation will reduce efficiency, thus intermediate results will not write into disk memory immediately. instead, those intermediate results will store in a buffer belonging to a Map node. Meanwhile, it will partition and group based on the original key and for every key it will be added a partition attribute value and stored together in a buffer. After Map shuffle will start the Reduce stage. It will re-extract intermediate results from buffers and do operations such as merge and reduce, gain final results and store them into HDFS. During shuffle it will execute frequent I/O operations and if datasets are big enough, the map task will also spill data to local disk buffers. Those will cause much more CPU calculation and CPU overhead, which affect execution efficiency.