Mini-Project 2

Do I need a data management system?

Nicolai Herforth

David Peter Mortensen

Ida Myrtue Søndermark

Frederik Gaasdal Jensen

Contents

1 Introduction

Data comes in many forms, where it might be either raw unstructured data, semi-structured or structured data in different formats as open-data-format or closed-format. This research paper examines the use of a data management systems like Apache Spark compared with the use of standalone Python code when processing data. Furthermore, it distinguishes between the two distinct data structures of Apache Spark (RDD and SQL), examining the pros and cons of each. However, the main focus is on performance and efficiency in accordance to the different methods regarding the use of a data management system, by answering four different data queries. The queries will be measured using different amounts of threads, with and without caching and loading different amounts of files.

2 Data Analysis Methods

The data analysis has been conducted with the use of different methods regarding the timing of the four different queries. Different time measurements have been collected from each individual query in relation to the specific performance. For measuring multi-threading efficiency on RDD and SQL respectively, timing experiments have been made for the individual queries, using x amount of threads for multi-threading. Moreover, there has been a focus specifically for RDD and SQL, where "with" and "without" caching was timed, in order to examine whether it has an effect on the processing. Furthermore, there has been a focus on testing the performance in regards to different amounts of files, and some of the experiments have been repeated to see if there were any exciting aspects worth noticing.

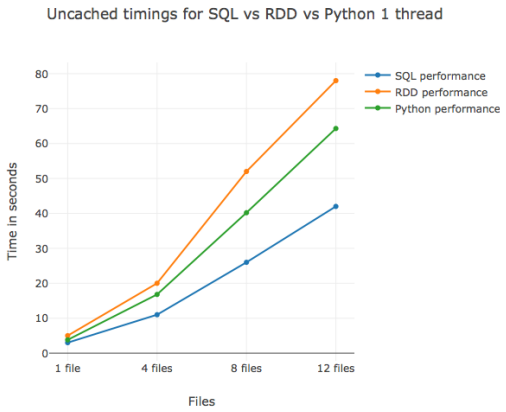
The data provided was conveniently split into 12 different files. This made it possible to do time experiments on different amounts of files in order to benchmark the performance of scalability of each method on various data sizes. The functionality of the different data management systems has also been examined with a focus on work-flow, limitations, and ease of use.

3 Results

The various timing experiments were performed by computing the results of the four queries. The actual results of the queries can be found in Appendix A.

The objective of this section is to present the different results in relation to the time experiments and test of efficiency based on a comparative analysis. Several results, in the form of plots regarding different performance parameters, have been attached in the appendix. Figure 1 displays the difference between the respective data management systems. The plot contains results on all systems with the use of only 1 thread attempted on various amounts of files. According to the plot, one can see that SQL has the most efficient performance, whereas it is possible to see that RDD is not quite as efficient and python lying in-between the two.

Figure 1



Applying caching makes a great difference in this case. In Appendices B and C, it is apparent that enabling caching for RDD slows down its performance remarkably because it could not hold the data in memory. However, SQL and Python make use of a much more efficient form of caching, making them far superior in this case. The remaining performance charts can be found in appendices D to G.

4 Interpretation of results

Standalone Python is as expected, a very good all-around approach to data management. Python by default caches all the data, taking up memory but performing multiple actions quickly. Multi-threading in Python is possible, although not as simple to implement as in Spark. Standalone Python is obviously the most flexible of the three since the management is not limited to a single data structure.

Apache Spark - RDD (Resilient Distributed Dataset) as a data structure is very simple, but scarce in regards to functionality. There is no direct column indexing, making it more suited for unstructured data instead of structured data. Caching is disabled by default due to the "lazy" nature of the data structure. By having it off, it is possible to work with large data sizes over multiple files, without overloading memory at the cost of every action taking longer to process. By

enabling caching, it is possible to store the data in memory, so each operation does not have to reevaluate the data on the second call. Multi-threading is by default enabled and can easily be customized.

Apache Spark - Data Frame (SQL) comes with a structured column based indexing syntax. This makes it very useful for structured data and makes indexing intuitive. Despite being the most organized structure, it still manages to almost be the most efficient. The Data Frame structure also comes with the same built-in multi-threading capabilities as RDD. Data Frames come with a bunch of handy and convenient usable commands for management.

Generally, it is very situational whether one should cache data. If performing multiple actions with a dataset small enough for the memory, it would be beneficial to cache. If this is not the case, then it would most likely be faster not to cache. Multi-threading in Spark generally increases performance drastically and in many cases when doubling the number of threads, it decreases the calculation time by up to 50%. Based on our performance results, Spark SQL seems to outshine Standalone Python, and in most cases it does. The Spark API offers many handy tools for big-data processing and its efficiency is outstanding. For very large data sizes, structured or unstructured, Spark becomes much preferable over Standalone Python. However, for daily tasks and projects with limited data sizes, Python would be perfectly fine, especially for unstructured data, where column indexing is not necessary.

5 Concluding Remarks

As a conclusion, one can say that Spark has superior performance over Python, in regards to our specific assignment. But when looking at the cached results, it is possible to see that Python actually out-performs SQL, resulting in a more optimal option to be used to answer multiple questions on the same dataset. However, if the data have to be loaded in every time for answering a question, SQL is a better option. It is also possible to see that SQL is a better option when it comes to structured data, but if it was allowed to use modules in Python, it could have the same functionality in the form of "Pandas". Finally, the experience was that it is a lot more convenient writing scripts in Python rather than coding live in Spark shell. The shell can be more convenient at times, but it is not as manageable.

6 Disclosure Statement

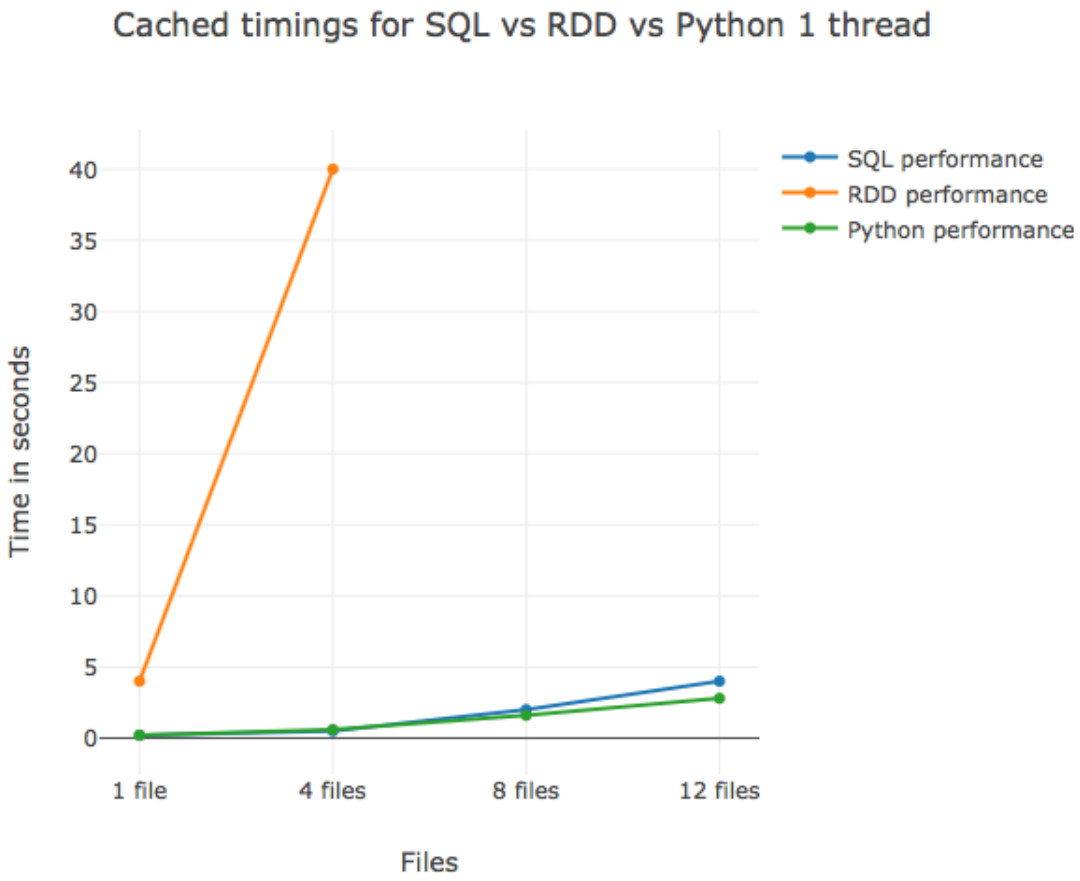
The workload have been equally distributed between the group members. As a group, we spent most of the time working together.

7 Appendix

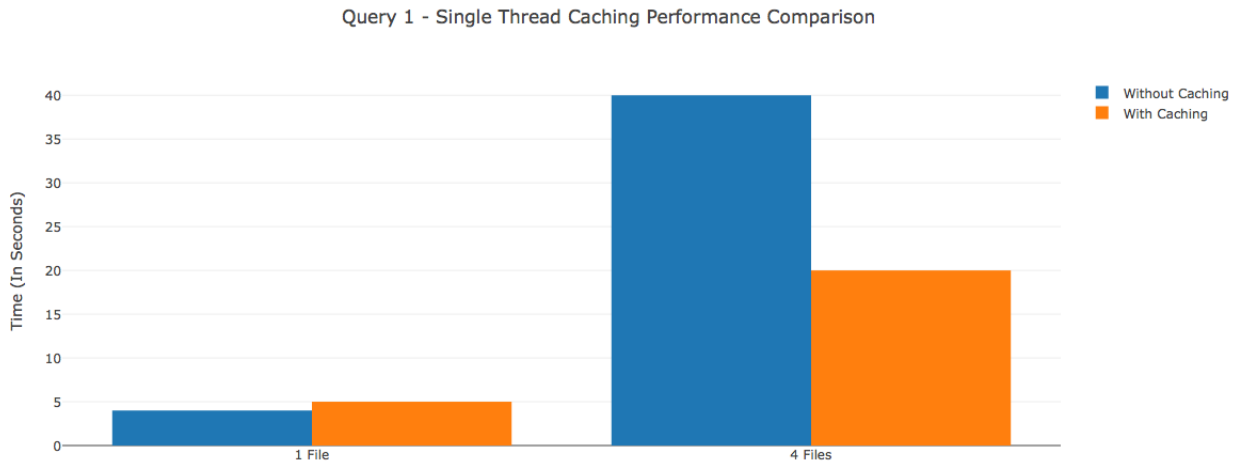
A Query Results

```
group9@pinar1:~$ python mini-2/chicago.py 1 chicago_taxi_drivers.csv query1
('time for reading in files:', 3.297914981842041)
1245712
QUERY 1
('The sum of the total payments is:', '18456147.90')
('time:', 0.3982889652252197)
group9@pinar1:~$ python mini-2/chicago.py 1 chicago_taxi_drivers.csv query2
('time for reading in files:', 3.288222074508667)
1245712
QUERY 2
{2: 4493.85, 3: 135672.66, 5: 1658.1, 6: 2457.5, 8: 1288846.98, 10: 286800.33, 11: 189.64, 12: 4672.87, 14: 2634.4, 15:
2496.9, 17: 1337.35, 19: 7389.72, 21: 3549.49, 26: 3504.5, 28: 2168.7, 31: 5762.33, 32: 3271.91, 35: 4928.8, 38: 3140.1,
42: 4329.6, 43: 257214.01, 44: 3676.1, 51: 14068.51, 55: 3498.85, 57: 1937.2, 62: 4198.53, 63: 4548.62, 65: 4602.1, 68:
39022.77, 72: 2845.37, 73: 2956.12, 74: 3838.72, 75: 4491.53, 79: 3729.73, 80: 3611.55, 82: 486707.87, 89: 4710.62, 90:
223657.53, 91: 1136.8, 92: 322973.0, 97: 6785.97, 99: 1660.15, 101: 1790053.22, 102: 4811.75, 103: 5132.98, 107:
3596232.56, 109: 1225856.68, 115: 2989.68, 116: 2328.2, 118: 2359.45}
9800941.9
('time:', 1.519244909286499)
group9@pinar1:~$ python mini-2/chicago.py 1 chicago_taxi_drivers.csv query3
('time for reading in files:', 3.303187847137451)
1245712
QUERY 3
('The sum of the total payments by cash is:', '8147721.81')
('time:', 0.308182954788208)
group9@pinar1:~$ python mini-2/chicago.py 1 chicago_taxi_drivers.csv query4
('time for reading in files:', 3.2859737873077393)
1245712
7666
QUERY 4
set(['GERRY', 'STYLES', 'MARBLES', "S'MORE", 'BLUE-GOTTI', 'VICTORY', 'ROLLIE'])
('number of different names:', 7)
('time:', 0.15135908126831055)
```

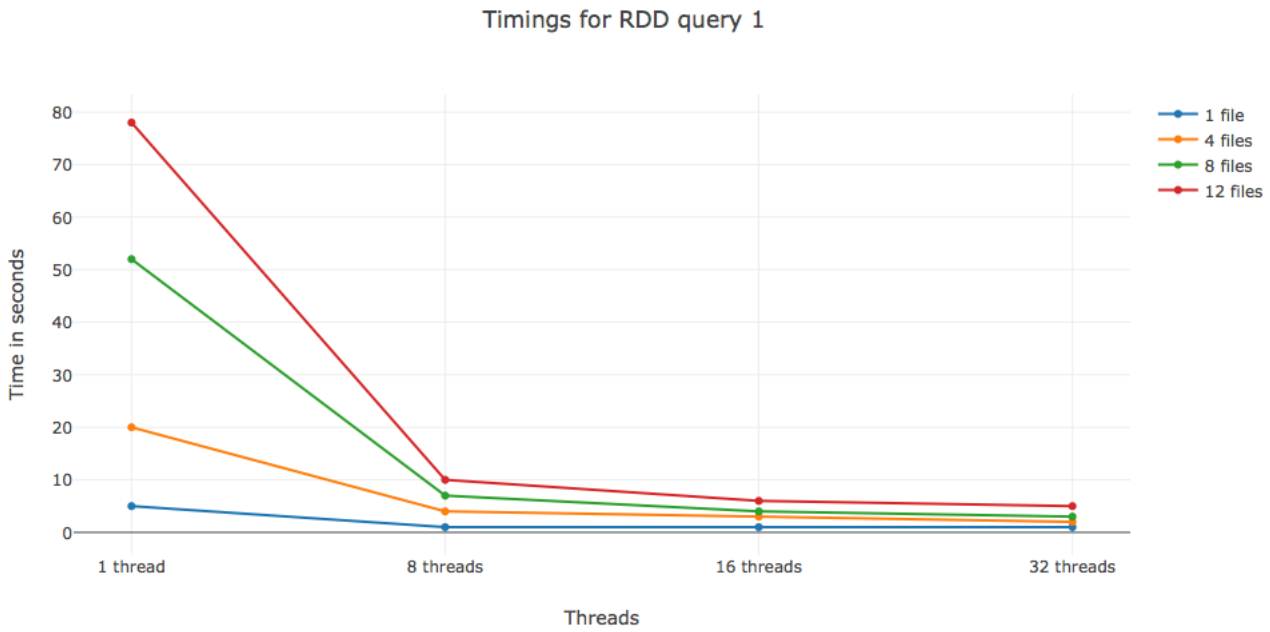
B Cached Timings

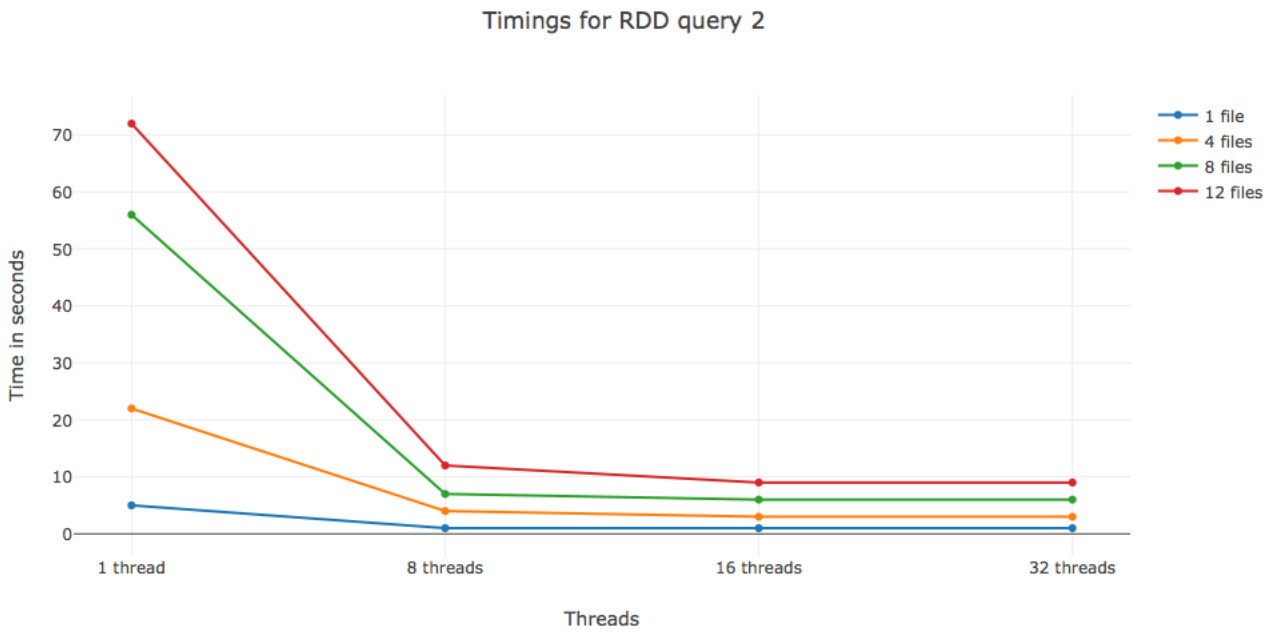


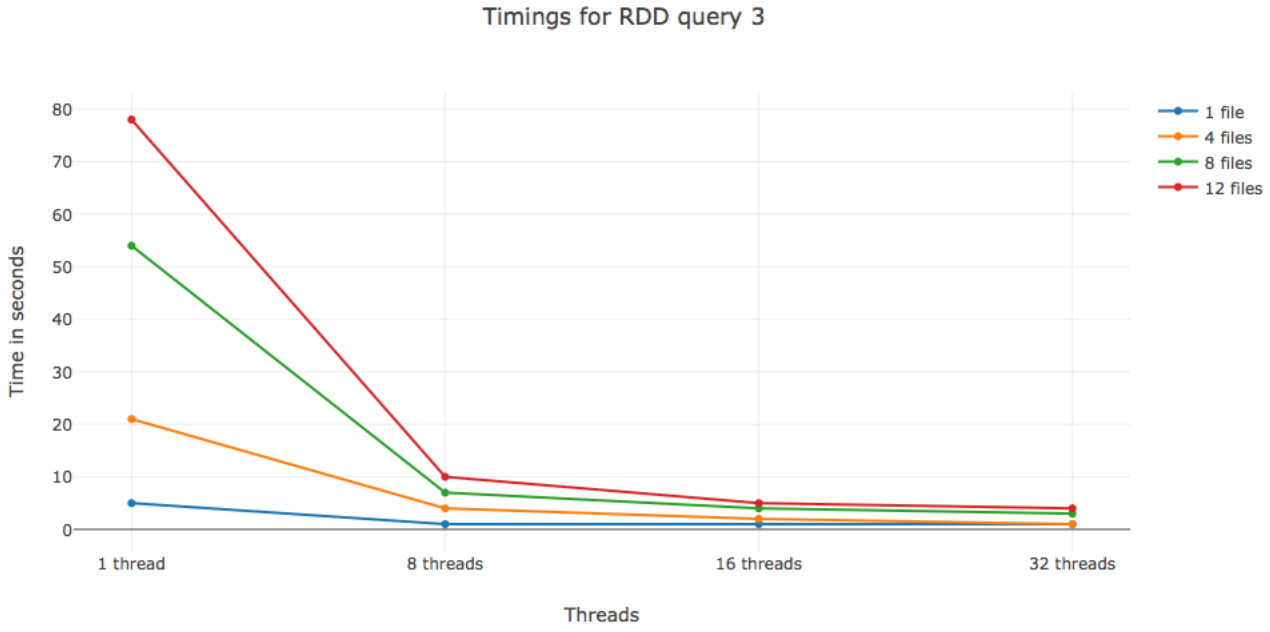
C Cached Timings RDD Comparison

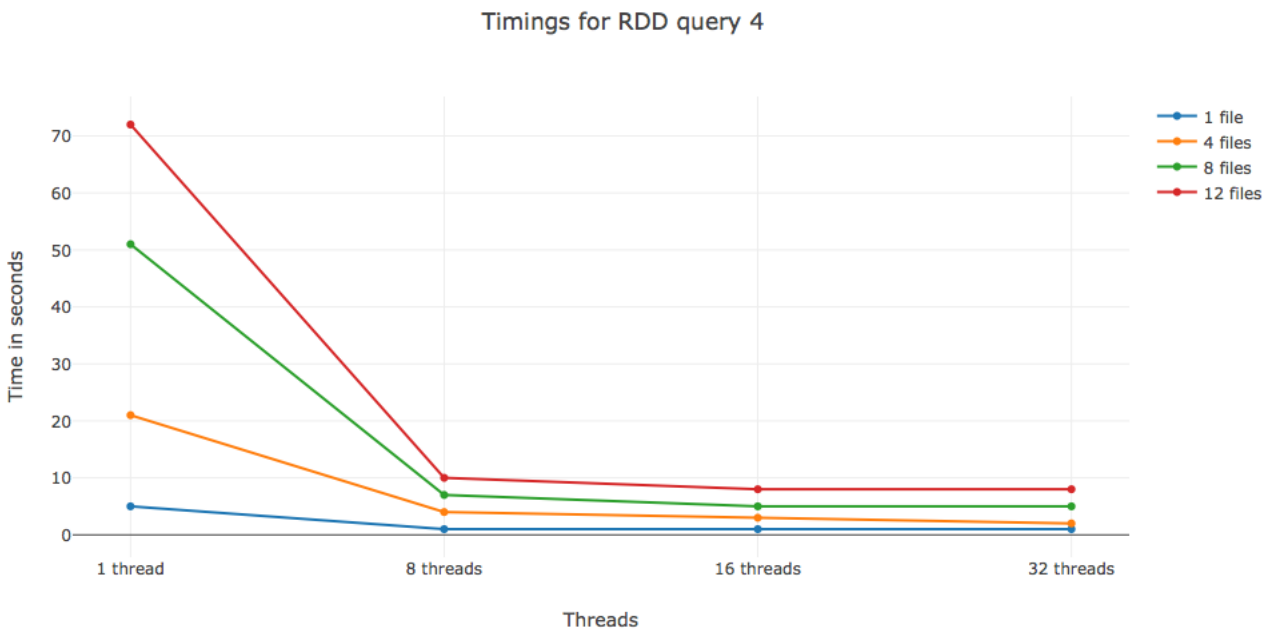


D Timings RDD Q1-4

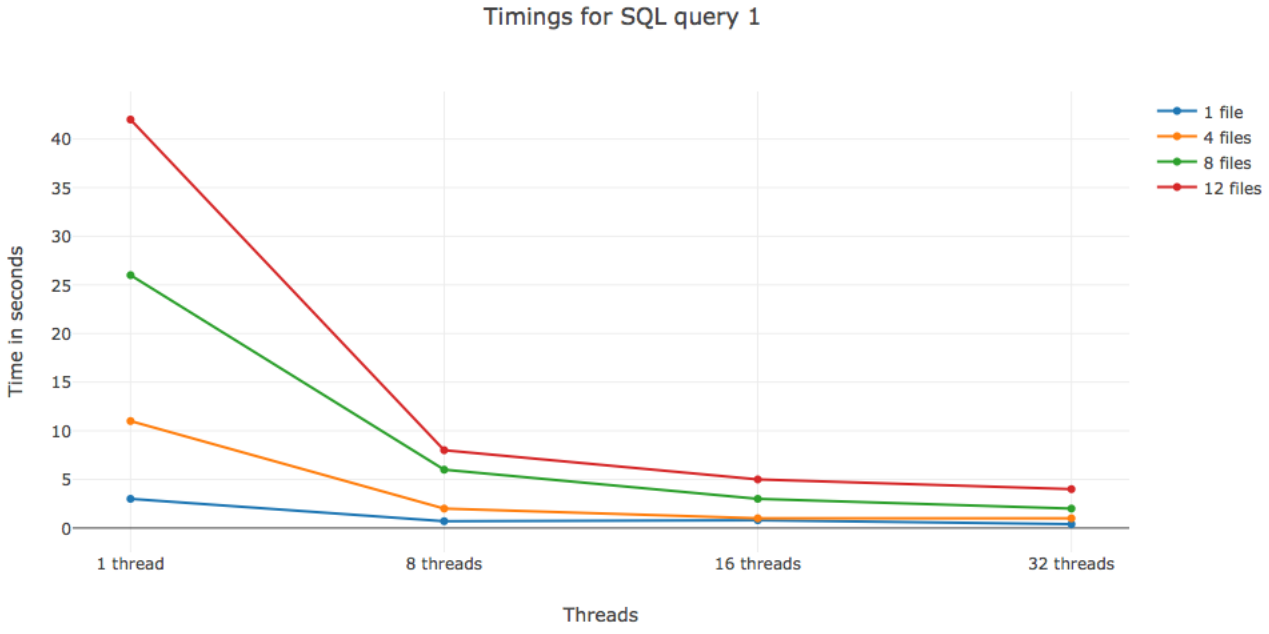


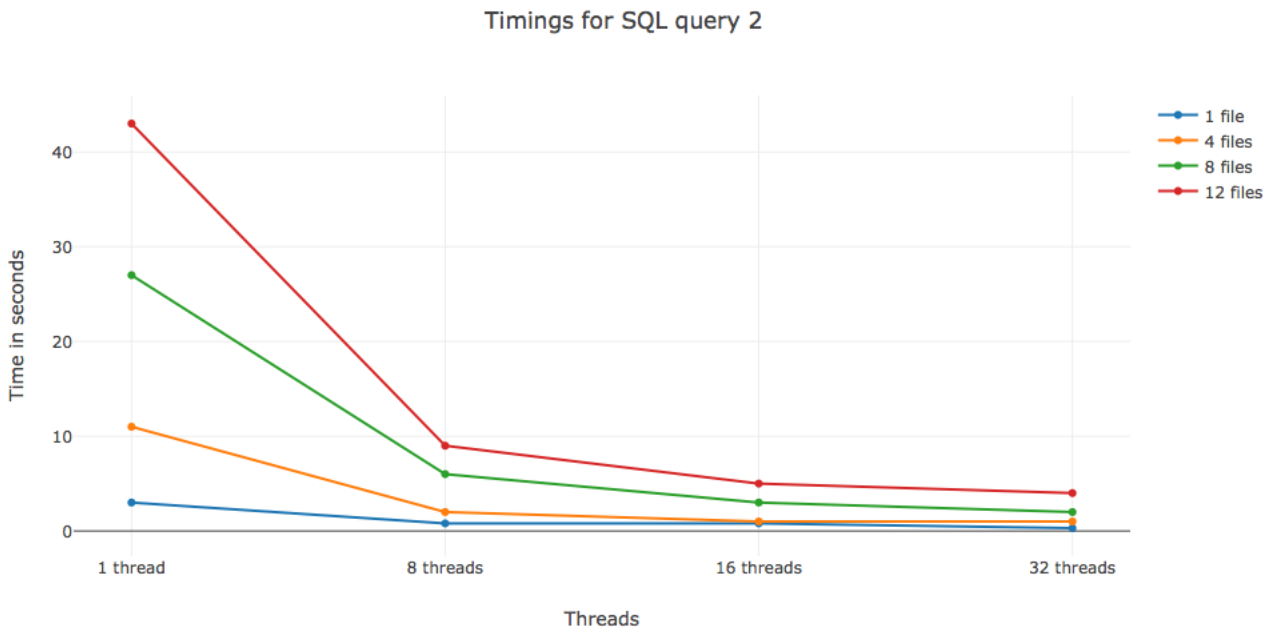

Timings for RDD query 2
Time in seconds
0
10
20
30
40
50
60
70
1 thread
8 threads
16 threads
32 threads
Threads
1 file
4 files
8 files
12 files

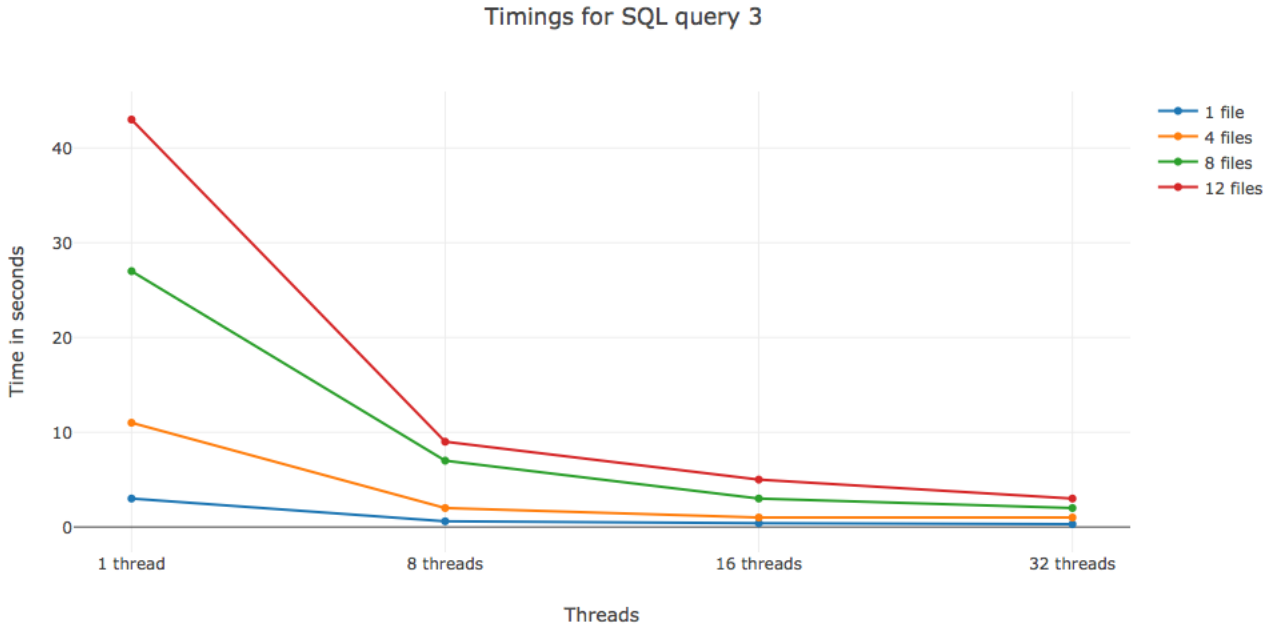

Timings for RDD query 3
Time in seconds
0
10
20
30
40
50
60
70
80
1 thread
8 threads
16 threads
32 threads
Threads
1 file
4 files
8 files
12 files

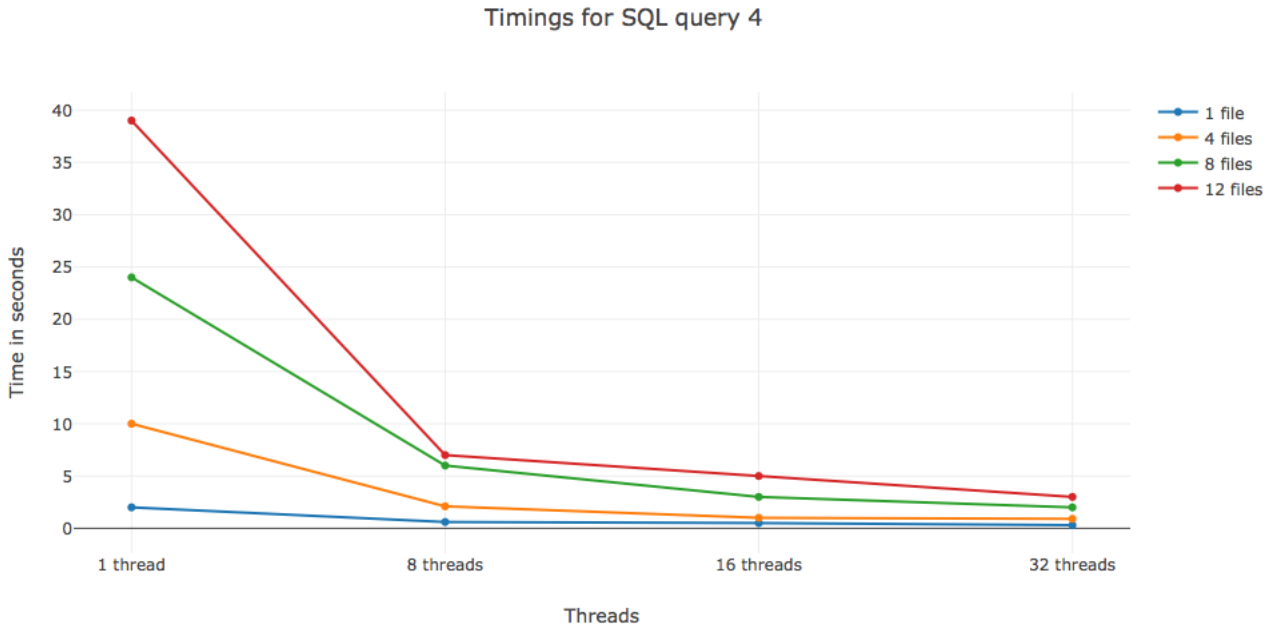


E Timings SQL Q1-4

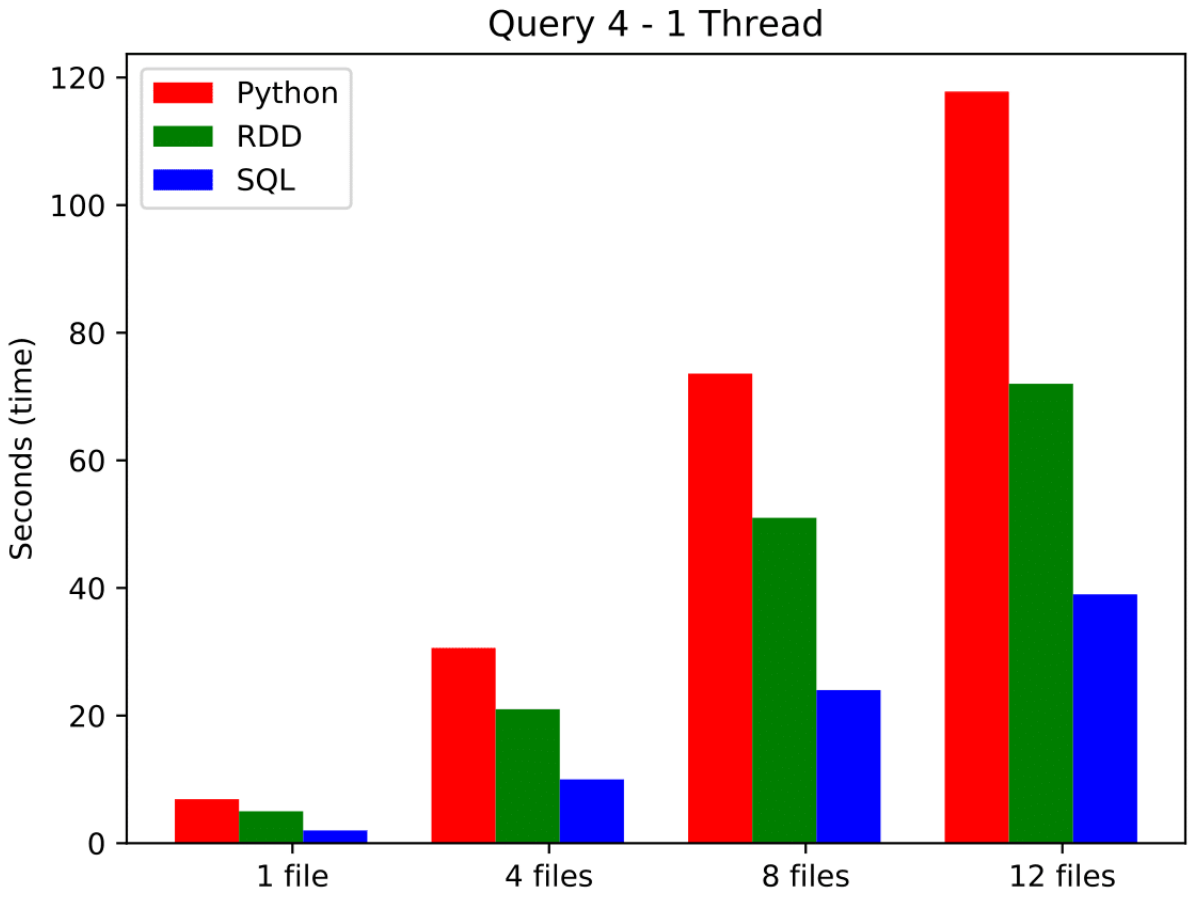



Timings for SQL query 2
Time in seconds
0
10
20
30
40
1 thread
8 threads
16 threads
32 threads
Threads
1 file
4 files
8 files
12 files

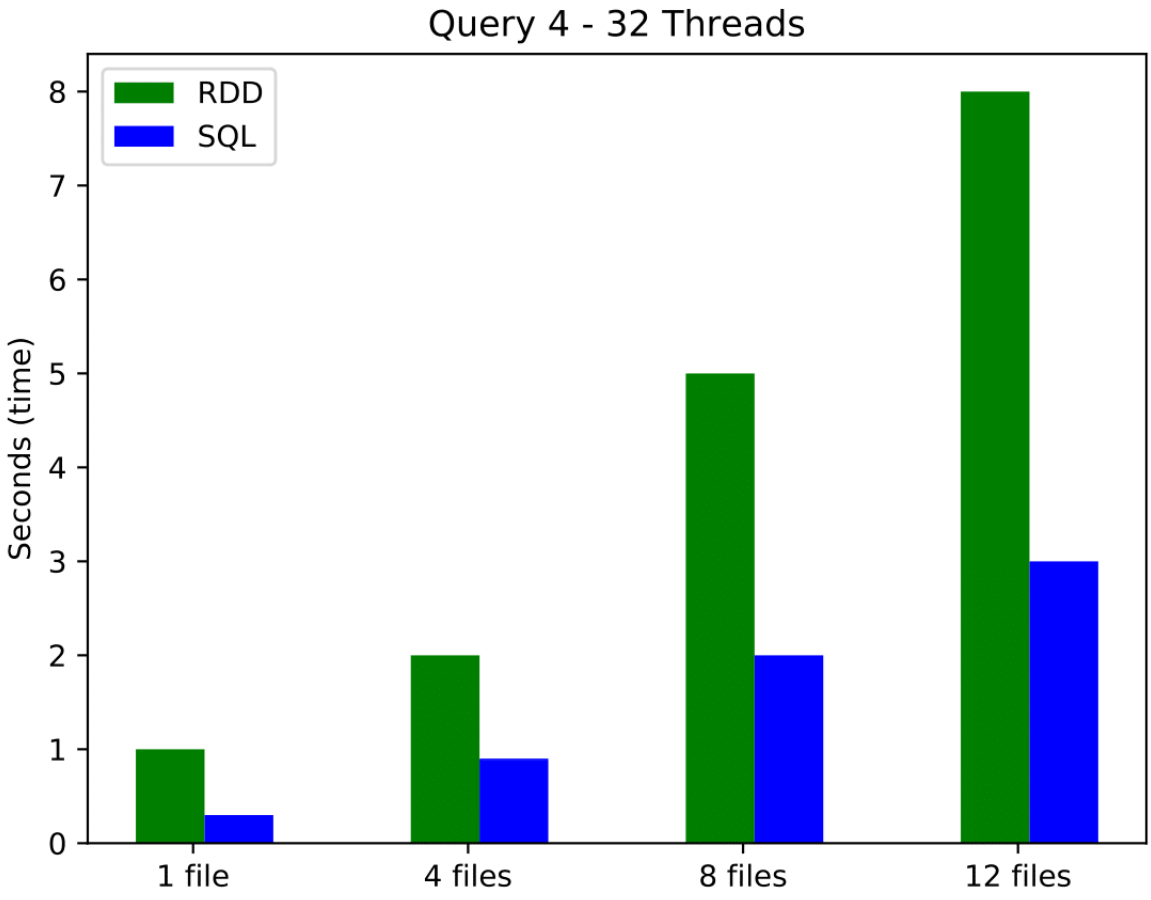

Timings for SQL query 3
Time in seconds
0
10
20
30
40
1 thread
8 threads
16 threads
32 threads
Threads
1 file
4 files
8 files
12 files


Timings for SQL query 4
Time in seconds
40
35
30
25
20
15
10
5
0
1 thread
8 threads
16 threads
32 threads
Threads
1 file
4 files
8 files
12 files

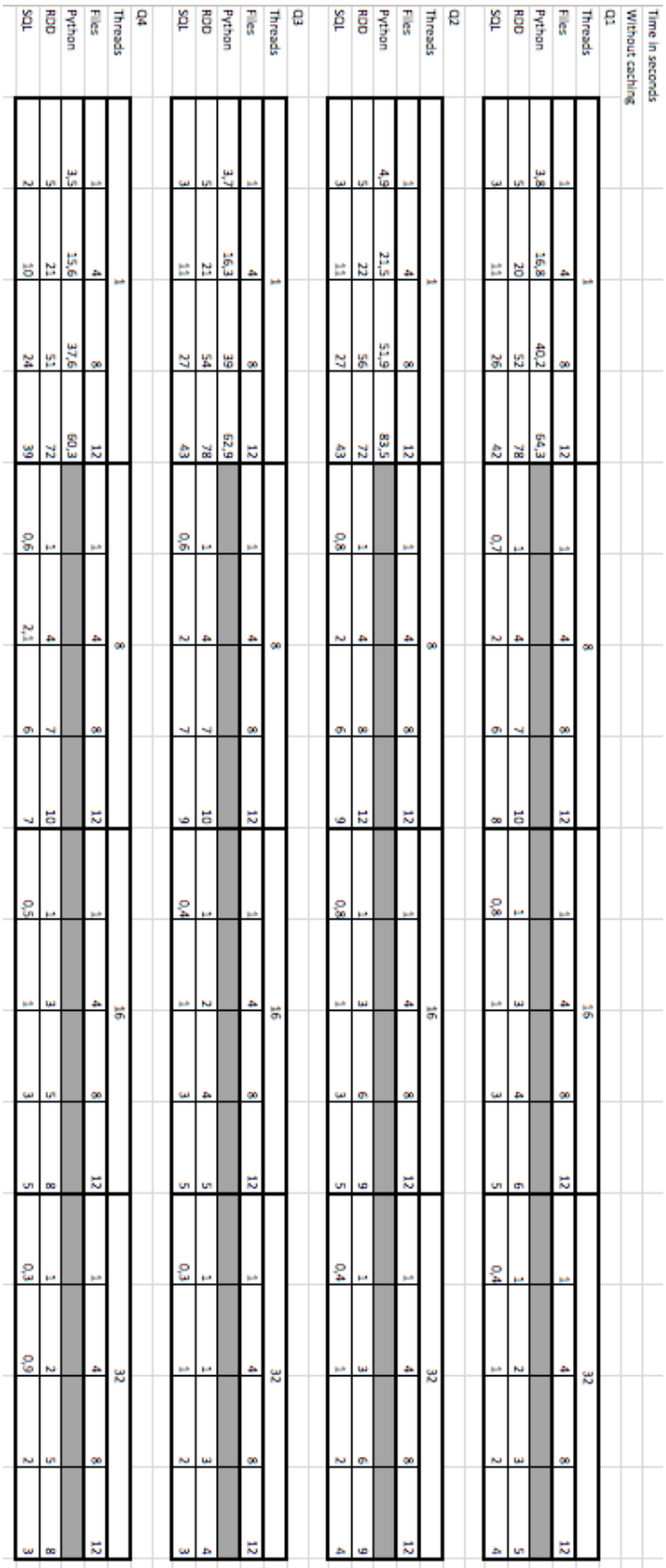
F Timings Query 4 on 1 thread



G Timings Query 4 on 32 threads



H Time measures without caching



Time in seconds

Without caching

Q1												
Threads	1			8			16			32		
Files	1	4	8	1	4	8	1	4	8	1	4	8
Python	3,8	16,8	40,2	64,3								
RDD	5	20	52	78	1	4	7	10	1	3	4	6
SQL	3	11	26	42	0,7	2	6	8	0,8	1	3	5

Q2												
Threads	1			8			16			32		
Files	1	4	8	12	1	4	8	12	1	4	8	12
Python	4,9	21,5	51,9	83,5								
RDD	5	22	56	72	1	4	8	12	1	3	6	9
SQL	3	11	27	43	0,8	2	6	9	0,8	1	3	5

Q3												
Threads	1			8			16			32		
Files	1	4	8	12	1	4	8	12	1	4	8	12
Python	3,7	16,3	39	62,9								
RDD	5	21	54	78	1	4	7	10	1	2	4	5
SQL	3	11	27	43	0,6	2	7	9	0,4	1	3	5

Q4												
Threads	1			8			16			32		
Files	1	4	8	12	1	4	8	12	1	4	8	12
Python	3,5	15,6	37,6	60,3								
RDD	5	21	51	72	1	4	7	10	1	3	5	8
SQL	2	10	24	39	0,6	2,1	6	7	0,5	1	3	5

Q1												
Threads	1			8			16			32		
Files	1	4	8	12	1	4	8	12	1	4	8	12
Python	3,8	16,8	40,2	64,3								
RDD	5	20	52	78	1	4	7	10	1	3	4	6
SQL	3	11	26	42	0,7	2	6	8	0,8	1	3	5

Q2												
Threads	1			8			16			32		
Files	1	4	8	12	1	4	8	12	1	4	8	12
Python	4,9	21,5	51,9	83,5								
RDD	5	22	56	72	1	4	8	12	1	3	6	9
SQL	3	11	27	43	0,8	2	6	9	0,8	1	3	5

Q3												
Threads	1			8			16			32		
Files	1	4	8	12	1	4	8	12	1	4	8	12
Python	3,7	16,3	39	62,9								
RDD	5	21	54	78	1	4	7	10	1	2	4	5
SQL	3	11	27	43	0,6	2	7	9	0,4	1	3	5

Q4												
Threads	1			8			16			32		
Files	1	4	8	12	1	4	8	12	1	4	8	12
Python	3,5	15,6	37,6	60,3								
RDD	5	21	51	72	1	4	7	10	1	3	5	8
SQL	2	10	24	39	0,6	2,1	6	7	0,5	1	3	5

I Time measures with caching

Time in seconds								
With caching								
Q1								
Threads	1				32			
Files	1	4	8	12	1	4	8	12
Python	0,4	1,8	4,2	6,8				
RDD	4	40	120	198	1	4	9	13
SQL	0,3	1	3	5	0,1	0,1	0,8	
Q2								
Threads	1				32			
Files	1	4	8	12	1	4	8	12
Python	1,5	6,5	15,9	26				
RDD	4	57	138	234	1	6	17	26
SQL	0,6	1	5	8	0,2	0,3		
Q3								
Threads	1				32			
Files	1	4	8	12	1	4	8	12
Python	0,3	1,3	3	5,4				
RDD	4	40	114	186	1	4	9	14
SQL	0,4	2	6	9	0,1	0,2	4	
Q4								
Threads	1				32			
Files	1	4	8	12	1	4	8	12
Python	0,2	0,6	1,6	2,8				
RDD	4	57	138	240	1	6	15	23
SQL	0,2	0,5	2	4	0,1	0,1	0,6	
Time in seconds								
Reading in files								
Threads	1				32			
Files	1	4	8	12	1	4	8	12
Python	3,4	15	36	57,5				
RDD	24	66	162	216	4	5	11	15
SQL	14	58	138	216	3	6	78	
Tasks								
Threads	1				32			
Files	1	4	8	12	1	4	8	12
RDD	5	20	45	71	5	20	45	71
SQL	2	5	12	19	32	32	33	33
SQL: From 8 files the space for cache is used up and moved to disk instead								
RDD: From 4 files the space for cache is used up and moved to disk instead								

J Time measures local

Time in seconds
Local

Q1

Threads	1		8	
Files	1	4	1	4
Python	6,2	90		
RDD	7	30	5	22
SQL	6	18	3	10

Q2

Threads	1		8	
Files	1	4	1	4
Python	7,9	100,6		
RDD	11	49	9	41
SQL	6	18	5	11

Q3

Threads	1		8	
Files	1	4	1	4
Python	6,3	90,8		
RDD	6	27	4	18
SQL	5	18	3	11

Q4

Threads	1		8	
Files	1	4	1	4
Python	5,7	88,4		
RDD	9	37	7	29
SQL	4	16	3	9

Python load 5,5 77,1

Time in seconds
Python no caching (second script)

Files	1	4	8	12
Q1	1,6	7	15	24,5
Q2	2,8	11,3	26,2	40,8
Q3	1,6	6,2	14	22,2
Q4	1,4	5,5	12,6	20,1

Time in seconds
Python no cahcing (second script) local

Files	1	4
Q1	2,7	12,5
Q2	4,3	17,9
Q3	2,6	10,7
Q4	2,2	9,3