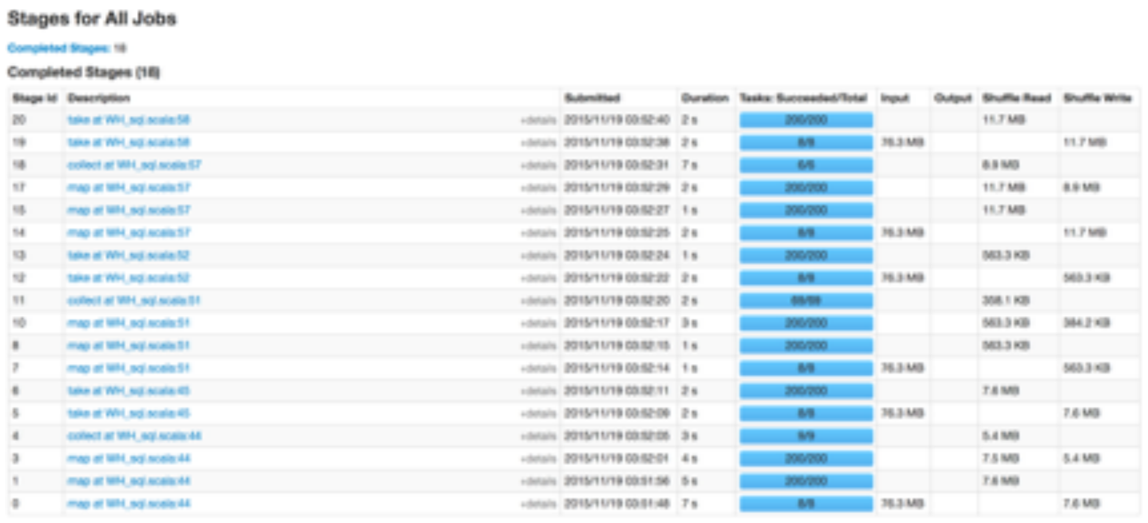
James Ziemba
CS 290
HW5

WhiteHouse Report:



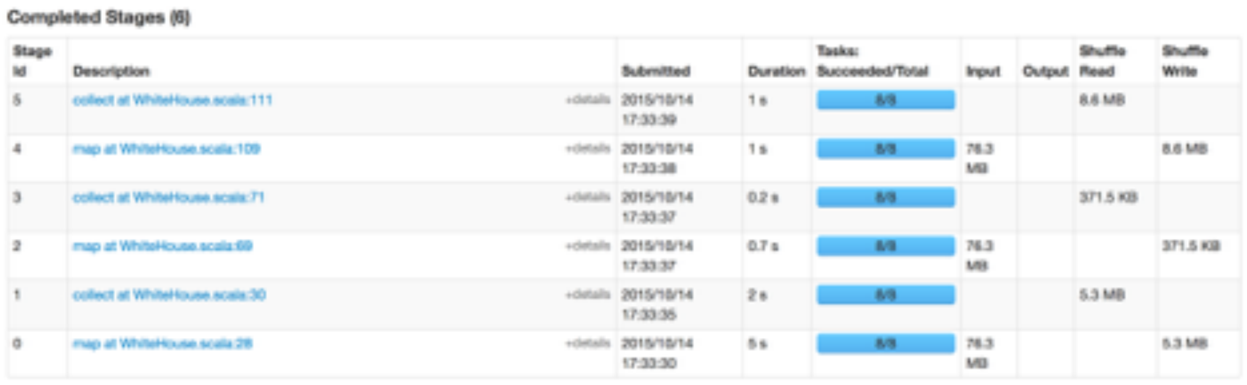
Stages for All Jobs

Completed Stages: 18

Completed Stages (18)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
20	take at WH_sql.scala:58 +details	2015/11/19 03:52:40	2 s	200/200			11.7 MB	
19	take at WH_sql.scala:58 +details	2015/11/19 03:52:38	2 s	8/8	76.3 MB			11.7 MB
18	collect at WH_sql.scala:57 +details	2015/11/19 03:52:31	7 s	6/6			8.9 MB	
17	map at WH_sql.scala:57 +details	2015/11/19 03:52:29	2 s	200/200			11.7 MB	8.9 MB
15	map at WH_sql.scala:57 +details	2015/11/19 03:52:27	1 s	200/200			11.7 MB	
14	map at WH_sql.scala:57 +details	2015/11/19 03:52:25	2 s	8/8	76.3 MB			11.7 MB
13	take at WH_sql.scala:52 +details	2015/11/19 03:52:24	1 s	200/200			563.3 KB	
12	take at WH_sql.scala:52 +details	2015/11/19 03:52:22	2 s	8/8	76.3 MB			563.3 KB
11	collect at WH_sql.scala:51 +details	2015/11/19 03:52:20	2 s	68/68			358.1 KB	
10	map at WH_sql.scala:51 +details	2015/11/19 03:52:17	3 s	200/200			563.3 KB	384.2 KB
8	map at WH_sql.scala:51 +details	2015/11/19 03:52:15	1 s	200/200			563.3 KB	
7	map at WH_sql.scala:51 +details	2015/11/19 03:52:14	1 s	8/8	76.3 MB			563.3 KB
6	take at WH_sql.scala:45 +details	2015/11/19 03:52:11	2 s	200/200			7.6 MB	
5	take at WH_sql.scala:45 +details	2015/11/19 03:52:09	2 s	8/8	76.3 MB			7.6 MB
4	collect at WH_sql.scala:44 +details	2015/11/19 03:52:05	3 s	9/9			5.4 MB	
3	map at WH_sql.scala:44 +details	2015/11/19 03:52:01	4 s	200/200			7.5 MB	5.4 MB
1	map at WH_sql.scala:44 +details	2015/11/19 03:51:56	5 s	200/200			7.6 MB	
0	map at WH_sql.scala:44 +details	2015/11/19 03:51:48	7 s	8/8	76.3 MB			7.6 MB

The above screenshot depicts the stages completed when my Spark SQL code executed. Below is a screenshot from when I originally ran it:



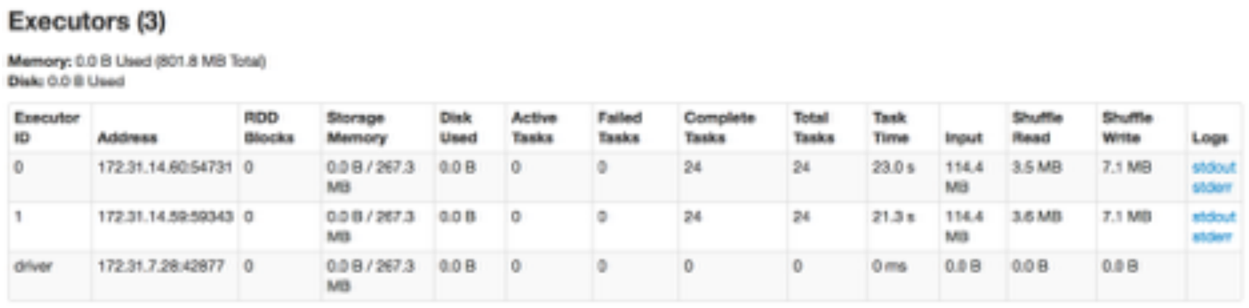
Completed Stages (6)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	collect at WhiteHouse.scala:111 +details	2015/10/14 17:33:39	1 s	8/8			8.6 MB	
4	map at WhiteHouse.scala:109 +details	2015/10/14 17:33:38	1 s	8/8	76.3 MB			8.6 MB
3	collect at WhiteHouse.scala:71 +details	2015/10/14 17:33:37	0.2 s	8/8			371.5 KB	
2	map at WhiteHouse.scala:69 +details	2015/10/14 17:33:37	0.7 s	8/8	76.3 MB			371.5 KB
1	collect at WhiteHouse.scala:30 +details	2015/10/14 17:33:35	2 s	8/8			5.3 MB	
0	map at WhiteHouse.scala:28 +details	2015/10/14 17:33:30	5 s	8/8	76.3 MB			5.3 MB

There are two glaring differences between these two runs: the Spark SQL code took significantly longer to run, which probably insinuates that the physical and/or logical plans were not as optimally constructed as they could have, and there are many more stages in the SQL run. Six of the extra stages are a result of me using an alternate way by which to print my results: I used a .take(10) to print the top 10 results (most common) of the results of the three queries. In my original code, I manually took out each of the top 10 tuples from the three desired results. The other extra stages are a result of a maximum number of tasks allowed at each stage (200). Each time I mapped the results of a query, three map stages were required to complete this operation.

As you can also see, there were significantly more tasks during each stage during the SQL run. I suspect that the efficiency of the plans selected to carry out the queries was not optimal. Also, the results of these queries could have been obtained with a simple reduceByKey function (as I did in my original code).

This screenshot is from when I ran my original WhiteHouse code. It shows the work completed by each slave:

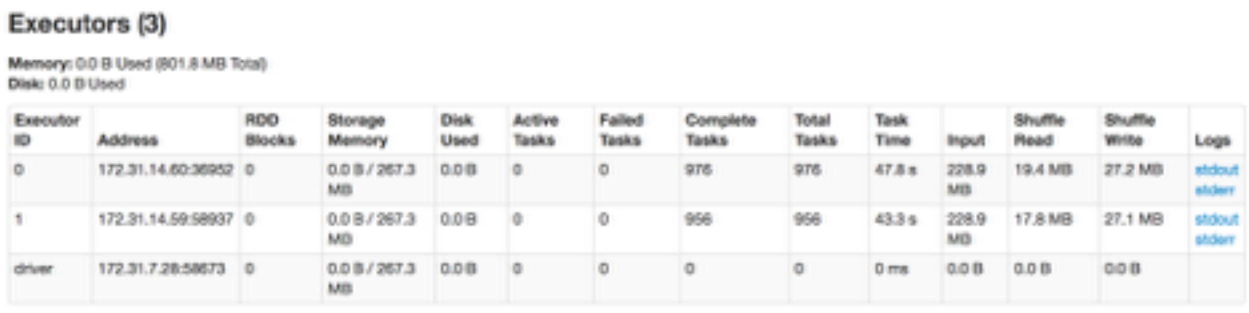


Executors (3)

Memory: 0.0 B Used (801.8 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
0	172.31.14.60:54731	0	0.0 B / 267.3 MB	0.0 B	0	0	24	24	23.0 s	114.4 MB	3.5 MB	7.1 MB	stdout stderr
1	172.31.14.59:59343	0	0.0 B / 267.3 MB	0.0 B	0	0	24	24	21.3 s	114.4 MB	3.6 MB	7.1 MB	stdout stderr
driver	172.31.7.28:42877	0	0.0 B / 267.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	

This one is when I used SQL:



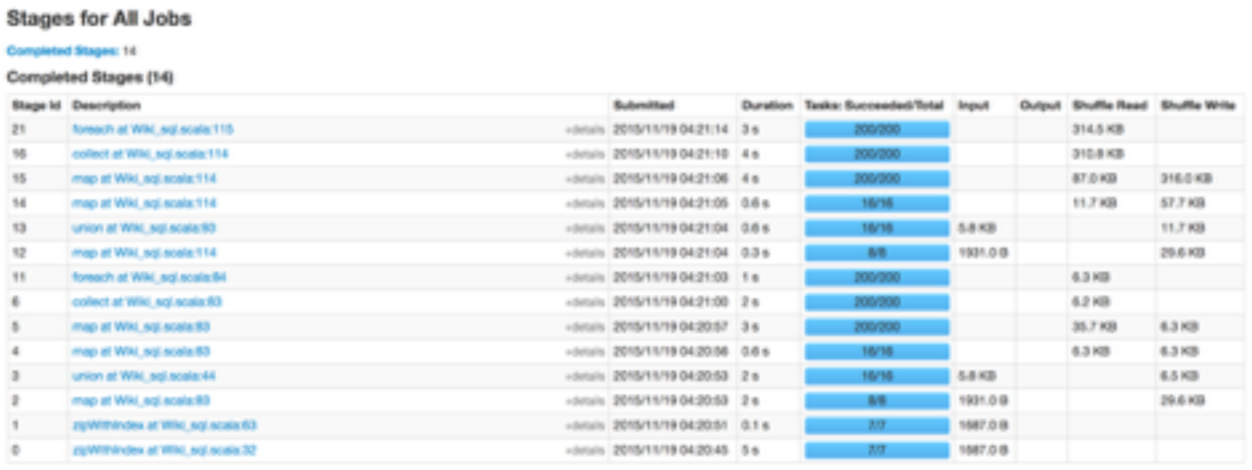
Executors (3)

Memory: 0.0 B Used (801.8 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
0	172.31.14.60:36952	0	0.0 B / 267.3 MB	0.0 B	0	0	976	976	47.8 s	228.9 MB	19.4 MB	27.2 MB	stdout stderr
1	172.31.14.59:58937	0	0.0 B / 267.3 MB	0.0 B	0	0	956	956	43.3 s	228.9 MB	17.8 MB	27.1 MB	stdout stderr
driver	172.31.7.28:58673	0	0.0 B / 267.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	

As you can see, there was significantly more data shuffled when I used SQL. I used a filter function when I wrote my original code, so I presume that my SQL query was not written efficiently as it could have been. Each partition completed significantly more tasks when I used SQL.

Wikipedia Report



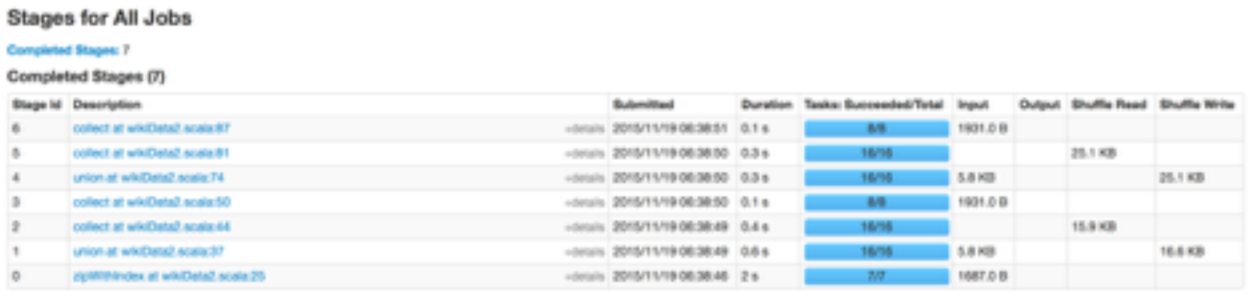
Stages for All Jobs

Completed Stages: 14

Completed Stages (14)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
21	foreach at Wiki_sql.scala:115	+details	2015/11/19 04:21:14	3 s	200/200			314.5 KB	
16	collect at Wiki_sql.scala:114	+details	2015/11/19 04:21:10	4 s	200/200			310.8 KB	
15	map at Wiki_sql.scala:114	+details	2015/11/19 04:21:06	4 s	200/200			87.0 KB	316.0 KB
14	map at Wiki_sql.scala:114	+details	2015/11/19 04:21:05	0.6 s	16/16			11.7 KB	57.7 KB
13	union at Wiki_sql.scala:93	+details	2015/11/19 04:21:04	0.6 s	16/16	5.8 KB			11.7 KB
12	map at Wiki_sql.scala:114	+details	2015/11/19 04:21:04	0.3 s	8/8	1931.0 B			29.6 KB
11	foreach at Wiki_sql.scala:84	+details	2015/11/19 04:21:03	1 s	200/200			6.3 KB	
6	collect at Wiki_sql.scala:83	+details	2015/11/19 04:21:00	2 s	200/200			6.2 KB	
5	map at Wiki_sql.scala:83	+details	2015/11/19 04:20:57	3 s	200/200			35.7 KB	6.3 KB
4	map at Wiki_sql.scala:83	+details	2015/11/19 04:20:56	0.6 s	16/16			6.3 KB	6.3 KB
3	union at Wiki_sql.scala:44	+details	2015/11/19 04:20:53	2 s	16/16	5.8 KB			6.5 KB
2	map at Wiki_sql.scala:83	+details	2015/11/19 04:20:53	2 s	8/8	1931.0 B			29.6 KB
1	zipWithIndex at Wiki_sql.scala:63	+details	2015/11/19 04:20:51	0.1 s	7/7	1687.0 B			
0	zipWithIndex at Wiki_sql.scala:32	+details	2015/11/19 04:20:45	5 s	7/7	1687.0 B			

The above screenshot gives a summary of all jobs performed from when I used SQL queries.



Stages for All Jobs

Completed Stages: 7

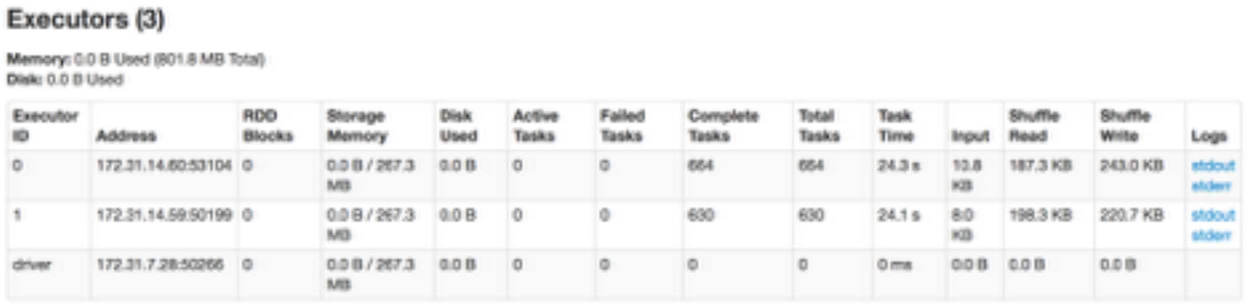
Completed Stages (7)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	collect at wikiData2.scala:87	+details	2015/11/19 06:38:51	0.1 s	8/8	1931.0 B			
5	collect at wikiData2.scala:81	+details	2015/11/19 06:38:50	0.3 s	16/16			25.1 KB	
4	union at wikiData2.scala:74	+details	2015/11/19 06:38:50	0.3 s	16/16	5.8 KB			25.1 KB
3	collect at wikiData2.scala:50	+details	2015/11/19 06:38:50	0.1 s	8/8	1931.0 B			
2	collect at wikiData2.scala:44	+details	2015/11/19 06:38:49	0.4 s	16/16			15.9 KB	
1	union at wikiData2.scala:37	+details	2015/11/19 06:38:49	0.6 s	16/16	5.8 KB			16.6 KB
0	zipWithIndex at wikiData2.scala:25	+details	2015/11/19 06:38:46	2 s	7/7	1687.0 B			

Again the two main differences between these two runs are the amount of tasks and how long the program took to run. Although the code with SQL did take longer to run, the discrepancy was not as significant as for WhiteHouse. I suspect that the method by which I extracted the answers, which I admit probably could have been better, is the reason for the large difference. Having to use a map function multiple times in order print the answers is a main reason that my SQL code took longer to run.

Similarly to WhiteHouse, the sheer amount of tasks caused the number of stages in the SQL run to be higher than in my original run. I also used a filter function throughout my original Wikipedia code, which definitely contributed to the faster run time.

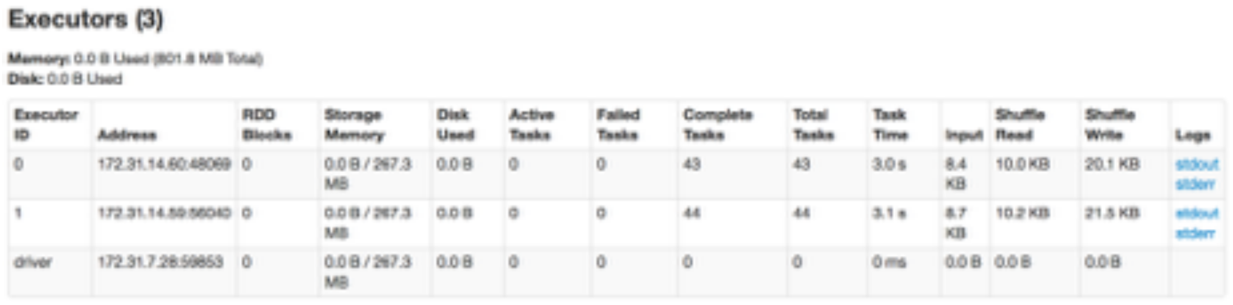
The top screenshot was taken from the SQL run and the bottom one was taken from my original run. Each slave had to read and write 10x as much data for the SQL run compared to my original one. Perhaps the SQL query did not filter out as much data it could have initially. In regards to how Spark executed the query, it filtered out some table values early in the physical plan (specified in the WHERE clause). This leads me to believe that the inefficiency of my method by which I obtained results from the query is the reason the SQL program took longer to execute. In addition, the .filter method I used in my original code could also have been a more efficient way by which to complete this problem.



Executors (3)

Memory: 0.0 B Used (801.8 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
0	172.31.14.60:53104	0	0.0 B / 267.3 MB	0.0 B	0	0	664	664	24.3 s	10.8 KB	187.3 KB	243.0 KB	stdout stderr
1	172.31.14.59:50199	0	0.0 B / 267.3 MB	0.0 B	0	0	630	630	24.1 s	8.0 KB	198.3 KB	220.7 KB	stdout stderr
driver	172.31.7.28:50266	0	0.0 B / 267.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	



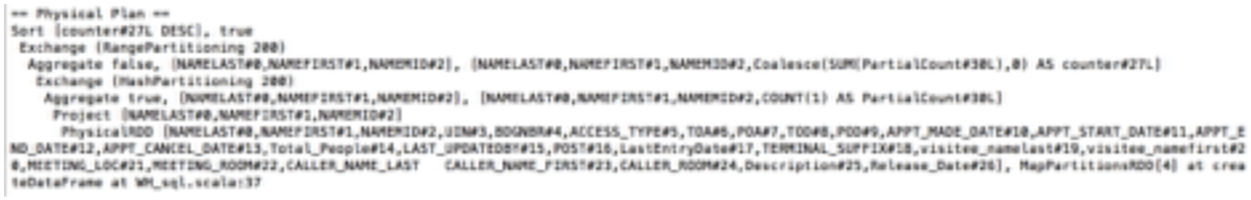
Executors (3)

Memory: 0.0 B Used (801.8 MB Total)
Disk: 0.0 B Used

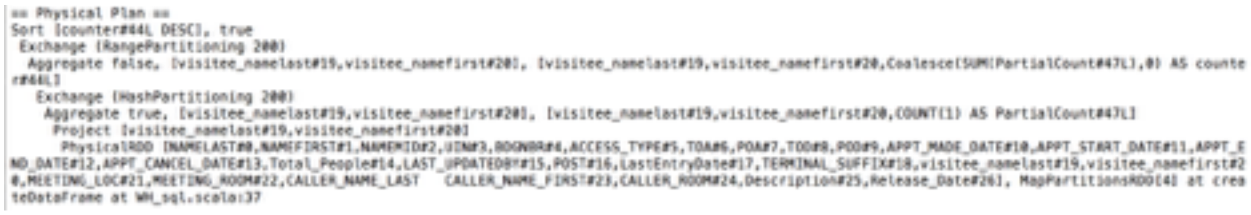
Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
0	172.31.14.60:48069	0	0.0 B / 267.3 MB	0.0 B	0	0	43	43	3.0 s	8.4 KB	10.0 KB	20.1 KB	stdout stderr
1	172.31.14.59:56040	0	0.0 B / 267.3 MB	0.0 B	0	0	44	44	3.1 s	8.7 KB	10.2 KB	21.5 KB	stdout stderr
driver	172.31.7.28:59853	0	0.0 B / 267.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	

WhiteHouse Plans

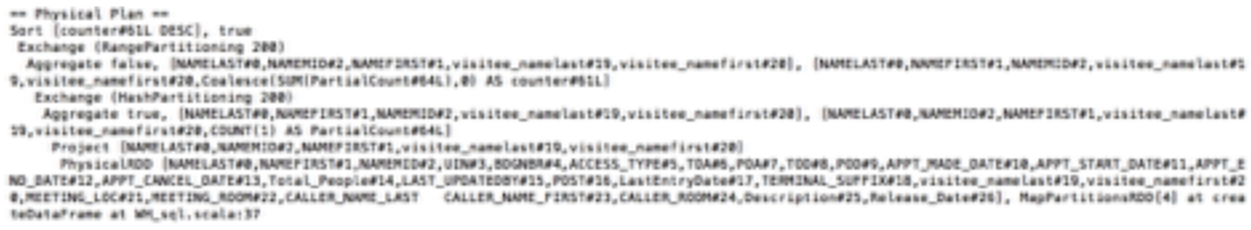
1.



```
== Physical Plan ==
Sort [counter#27L DESC], true
 Exchange (RangePartitioning 200)
  Aggregate false, [NAMELAST#0,NAMEFIRST#1,NAMEMID#2], [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,Coalesce(SUM(PartialCount#30L),0) AS counter#27L]
   Exchange (HashPartitioning 200)
    Aggregate true, [NAMELAST#0,NAMEFIRST#1,NAMEMID#2], [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,COUNT(1) AS PartialCount#30L]
     Project [NAMELAST#0,NAMEFIRST#1,NAMEMID#2]
      PhysicalRDD [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,UIN#3,BDGNBR#4,ACCESS_TYPE#5,TOA#6,POA#7,TOD#8,POD#9,APPT_MADE_DATE#10,APPT_START_DATE#11,APPT_E
ND_DATE#12,APPT_CANCEL_DATE#13,Total_People#14,LAST_UPDATEDBY#15,POST#16,LastEntryDate#17,TERMINAL_SUFFIX#18,visitee_namelast#19,visitee_namefirst#2
0,MEETING_LOC#21,MEETING_ROOM#22,CALLER_NAME_LAST   CALLER_NAME_FIRST#23,CALLER_ROOM#24,Description#25,Release_Date#26], MapPartitionsRDD[4] at crea
teDataFrame at WH_sql.scala:37
```



```
== Physical Plan ==
Sort [counter#44L DESC], true
 Exchange (RangePartitioning 200)
  Aggregate false, [visitee_namelast#19,visitee_namefirst#20], [visitee_namelast#19,visitee_namefirst#20,Coalesce(SUM(PartialCount#47L),0) AS counte
r#44L]
   Exchange (HashPartitioning 200)
    Aggregate true, [visitee_namelast#19,visitee_namefirst#20], [visitee_namelast#19,visitee_namefirst#20,COUNT(1) AS PartialCount#47L]
     Project [visitee_namelast#19,visitee_namefirst#20]
      PhysicalRDD [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,UIN#3,BDGNBR#4,ACCESS_TYPE#5,TOA#6,POA#7,TOD#8,POD#9,APPT_MADE_DATE#10,APPT_START_DATE#11,APPT_E
ND_DATE#12,APPT_CANCEL_DATE#13,Total_People#14,LAST_UPDATEDBY#15,POST#16,LastEntryDate#17,TERMINAL_SUFFIX#18,visitee_namelast#19,visitee_namefirst#2
0,MEETING_LOC#21,MEETING_ROOM#22,CALLER_NAME_LAST   CALLER_NAME_FIRST#23,CALLER_ROOM#24,Description#25,Release_Date#26], MapPartitionsRDD[4] at crea
teDataFrame at WH_sql.scala:37
```



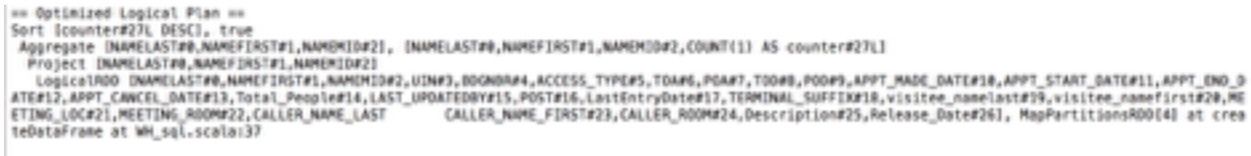
```
== Physical Plan ==
Sort [counter#61L DESC], true
 Exchange (RangePartitioning 200)
  Aggregate false, [NAMELAST#0,NAMEMID#2,NAMEFIRST#1,visitee_namelast#19,visitee_namefirst#20], [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,visitee_namelast#1
9,visitee_namefirst#20,Coalesce(SUM(PartialCount#64L),0) AS counter#61L]
   Exchange (HashPartitioning 200)
    Aggregate true, [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,visitee_namelast#19,visitee_namefirst#20], [NAMELAST#0,NAMEMID#2,NAMEFIRST#1,visitee_namelast#
19,visitee_namefirst#20,COUNT(1) AS PartialCount#64L]
     Project [NAMELAST#0,NAMEMID#2,NAMEFIRST#1,visitee_namelast#19,visitee_namefirst#20]
      PhysicalRDD [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,UIN#3,BDGNBR#4,ACCESS_TYPE#5,TOA#6,POA#7,TOD#8,POD#9,APPT_MADE_DATE#10,APPT_START_DATE#11,APPT_E
ND_DATE#12,APPT_CANCEL_DATE#13,Total_People#14,LAST_UPDATEDBY#15,POST#16,LastEntryDate#17,TERMINAL_SUFFIX#18,visitee_namelast#19,visitee_namefirst#2
0,MEETING_LOC#21,MEETING_ROOM#22,CALLER_NAME_LAST   CALLER_NAME_FIRST#23,CALLER_ROOM#24,Description#25,Release_Date#26], MapPartitionsRDD[4] at crea
teDataFrame at WH_sql.scala:37
```

The above three screenshots depict the three physical plans for the WhiteHouse queries that I ran. Because I used the same framework for each query and only changed around what the SELECT clause chose, the physical plans are very similar. Both hash partitioning and range partitioning was used. The range partitioning was used right before the output of all three queries were sorted. An aggregate function was used twice because of the count aspect of the query.

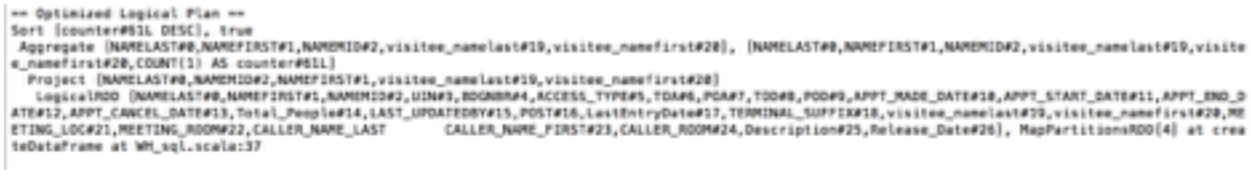
Below is an example of the queries that I used to solve these 3 problems:

SELECT NAMELAST,NAMEFIRST,NAMEMID,COUNT(*)
AS counter
FROM visitors
GROUP BY NAMELAST,NAMEFIRST,NAMEMID
ORDER BY counter DESC

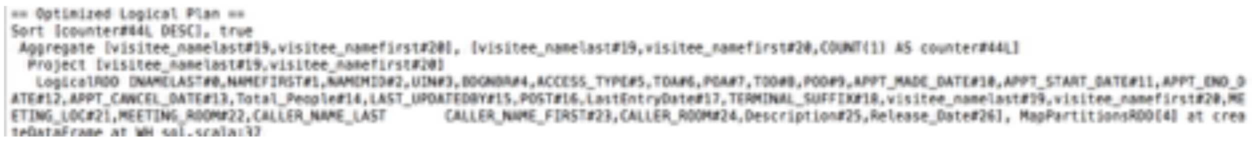
The only part of this query that changed among the three queries was what was SELECTed.



```
== Optimized Logical Plan ==
Sort [counter#27L DESC], true
 Aggregate [NAMELAST#0,NAMEFIRST#1,NAMEMID#2], [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,COUNT(1) AS counter#27L]
  Project [NAMELAST#0,NAMEFIRST#1,NAMEMID#2]
   LogicalRDD [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,UIN#3,BDGNBR#4,ACCESS_TYPE#5,TOA#6,POA#7,TOD#8,POD#9,APPT_MADE_DATE#10,APPT_START_DATE#11,APPT_END_D
ATE#12,APPT_CANCEL_DATE#13,Total_People#14,LAST_UPDATEDBY#15,POST#16,LastEntryDate#17,TERMINAL_SUFFIX#18,visitee_namelast#19,visitee_namefirst#20,ME
ETING_LOC#21,MEETING_ROOM#22,CALLER_NAME_LAST       CALLER_NAME_FIRST#23,CALLER_ROOM#24,Description#25,Release_Date#26], MapPartitionsRDD[4] at crea
teDataFrame at WH_sql.scala:37
```



```
== Optimized Logical Plan ==
Sort [counter#61L DESC], true
 Aggregate [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,visitee_namelast#19,visitee_namefirst#20], [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,visitee_namelast#19,visite
e_namefirst#20,COUNT(1) AS counter#61L]
  Project [NAMELAST#0,NAMEMID#2,NAMEFIRST#1,visitee_namelast#19,visitee_namefirst#20]
   LogicalRDD [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,UIN#3,BDGNBR#4,ACCESS_TYPE#5,TOA#6,POA#7,TOD#8,POD#9,APPT_MADE_DATE#10,APPT_START_DATE#11,APPT_END_D
ATE#12,APPT_CANCEL_DATE#13,Total_People#14,LAST_UPDATEDBY#15,POST#16,LastEntryDate#17,TERMINAL_SUFFIX#18,visitee_namelast#19,visitee_namefirst#20,ME
ETING_LOC#21,MEETING_ROOM#22,CALLER_NAME_LAST       CALLER_NAME_FIRST#23,CALLER_ROOM#24,Description#25,Release_Date#26], MapPartitionsRDD[4] at crea
teDataFrame at WH_sql.scala:37
```



```
== Optimized Logical Plan ==
Sort [counter#44L DESC], true
 Aggregate [visitee_namelast#19,visitee_namefirst#20], [visitee_namelast#19,visitee_namefirst#20,COUNT(1) AS counter#44L]
  Project [visitee_namelast#19,visitee_namefirst#20]
   LogicalRDD [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,UIN#3,BDGNBR#4,ACCESS_TYPE#5,TOA#6,POA#7,TOD#8,POD#9,APPT_MADE_DATE#10,APPT_START_DATE#11,APPT_END_D
ATE#12,APPT_CANCEL_DATE#13,Total_People#14,LAST_UPDATEDBY#15,POST#16,LastEntryDate#17,TERMINAL_SUFFIX#18,visitee_namelast#19,visitee_namefirst#20,ME
ETING_LOC#21,MEETING_ROOM#22,CALLER_NAME_LAST        CALLER_NAME_FIRST#23,CALLER_ROOM#24,Description#25,Release_Date#26], MapPartitionsRDD[4] at crea
teDataFrame at WH_sql.scala:37
```

Above are the three optimized logical plans for each of the three WhiteHouse queries. I will present one alternative logical plan because the optimized logical plans above are all identical besides what is selected.

In my alternate logical plan, I would change the location of the projection of what was being selected. In this alternate plan, the key pieces of data (NAMELAST, NAMEFIRST, NAMEMID, visitee_namelast, visitee_namefirst) would not be projected until directly after the sort action occurred. Because we would be carrying more data throughout the duration of the operation, more data would have to be shuffled throughout each operation, which would considerably slow down the execution of the query. Because of this fact alone, the logical plan selected by Spark would certainly be the better one to select. Although the cost in terms of getNext calls would not change (because the number of table entries would not change), cost in terms of amount of data moved throughout my query would be higher.

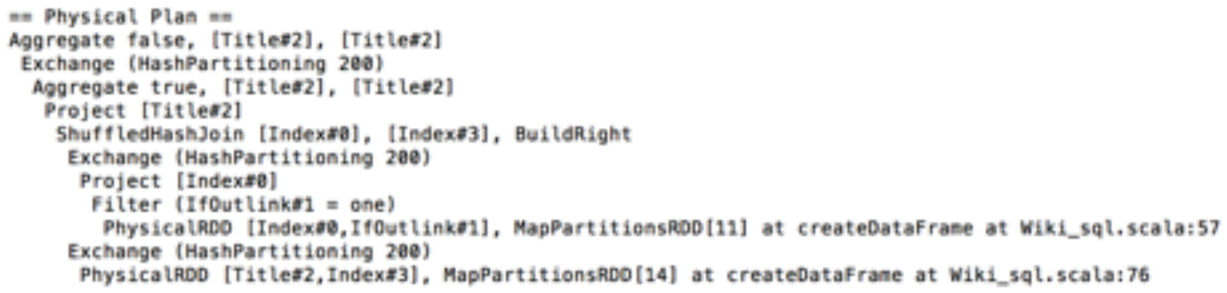
My Alternate Plan
```
== Optimized Logical Plan ==
Project [NAMELAST#0,NAMEFIRST#1,NAMEMID#2]
Sort [counter#27L DESC], true
 Aggregate [NAMELAST#0,NAMEFIRST#1,NAMEMID#2],
[NAMELAST#0,NAMEFIRST#1,NAMEMID#2,COUNT(1) AS counter#27L]
   LogicalRDD […schema…]
```

As I explained above, the physical plans, like the logical plans, are all identical besides what was being selected. Similarly with my alternate logical plan, I could choose to project the desired data at a time other than the beginning of the execution of the query. In this case, if I waited to do so after this line (so “above” it in the screenshot):

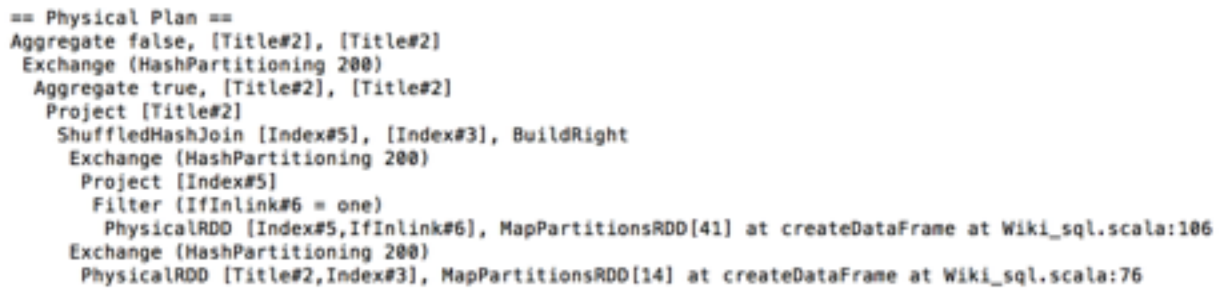
```
Project [NAMELAST#0,NAMEFIRST#1,NAMEMID#2] <—— change
Aggregate true, [NAMELAST#0,NAMEFIRST#1,NAMEMID#2],
     [NAMELAST#0,NAMEFIRST#1,NAMEMID#2,COUNT(1) AS PartialCount#30L]
```
then I could limit the “damage” of not projecting just the relevant data at the beginning of the plan. If I carry out with the rest of the query as I have it set up ((namelast,namefirst,namemid),1) would be the only pieces of data that would be joined, which would not be as optimal as the plan chosen by Spark, but certainly a better idea than my proposed logical plan.

My Alternate Plan:
```
== Physical Plan ==
Sort [counter#27L DESC], true
 Exchange (RangePartitioning 200)
Project [NAMELAST#0,NAMEFIRST#1,NAMEMID#2]
  Aggregate false, [NAMELAST#0,NAMEFIRST#1,NAMEMID#2],
[NAMELAST#0,NAMEFIRST#1,NAMEMID#2,Coalesce(SUM(PartialCount#30L),0) AS
counter#27L]
   Exchange (HashPartitioning 200)
    Aggregate true, [NAMELAST#0,NAMEFIRST#1,NAMEMID#2],
[NAMELAST#0,NAMEFIRST#1,NAMEMID#2,COUNT(1) AS PartialCount#30L]
      PhysicalRDD[…schema…]
```

Wikipedia Plans



```
== Physical Plan ==
Aggregate false, [Title#2], [Title#2]
 Exchange (HashPartitioning 200)
  Aggregate true, [Title#2], [Title#2]
   Project [Title#2]
    ShuffledHashJoin [Index#0], [Index#3], BuildRight
     Exchange (HashPartitioning 200)
      Project [Index#0]
       Filter (IfOutlink#1 = one)
        PhysicalRDD [Index#0,IfOutlink#1], MapPartitionsRDD[11] at createDataFrame at Wiki_sql.scala:57
     Exchange (HashPartitioning 200)
      PhysicalRDD [Title#2,Index#3], MapPartitionsRDD[14] at createDataFrame at Wiki_sql.scala:76
```



```
== Physical Plan ==
Aggregate false, [Title#2], [Title#2]
 Exchange (HashPartitioning 200)
  Aggregate true, [Title#2], [Title#2]
   Project [Title#2]
    ShuffledHashJoin [Index#5], [Index#3], BuildRight
     Exchange (HashPartitioning 200)
      Project [Index#5]
       Filter (IfInlink#6 = one)
        PhysicalRDD [Index#5,IfInlink#6], MapPartitionsRDD[41] at createDataFrame at Wiki_sql.scala:106
     Exchange (HashPartitioning 200)
      PhysicalRDD [Title#2,Index#3], MapPartitionsRDD[14] at createDataFrame at Wiki_sql.scala:76
```

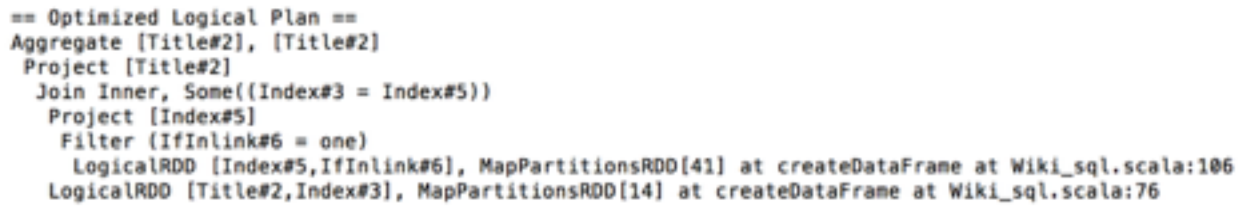
Above are the two physical plans from the two Wikipedia queries. Similarly to my WhiteHouse queries, my Wikipedia ones were very similar, with only a few changes.

```
SELECT titles.Title
FROM no_outlinks,titles
WHERE (no_outlinks.IfOutlink = 'one' AND titles.Index = no_outlinks.Index)
GROUP BY titles.Title
```

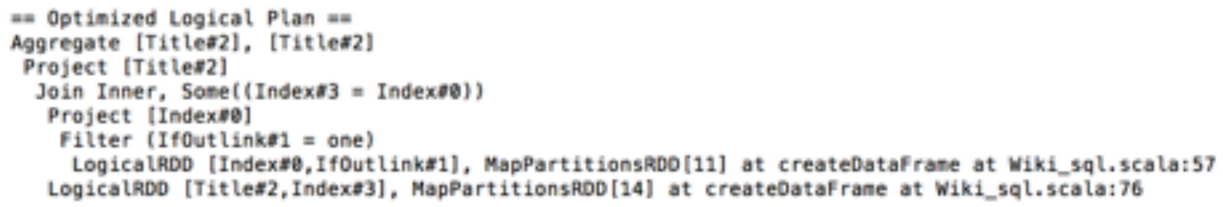
```
SELECT titles.Title
FROM no_inlinks,titles
WHERE (no_inlinks.IfInlink = 'one' AND titles.Index = no_inlinks.Index)
GROUP BY titles.Title
```

As you can see, two tables on which I am trying to join (no_outlinks and no_inlinks) are formatted in such a way where one of the attributes of its elements (IfOutlink/IfInlink) is set to “one” if it conforms to the goal of the query (has no inlinks or has no outlinks).

An initial filter of no_outlinks or no_inlinks gets rid of all entries where IfOutlink/IfInlink != “one.” This helps eliminate a large number of pages because many both have outlinks and inlinks. This greatly reduces the amount of data that has to be shuffled subsequently. The indexes from the no_outlinks/no_inlinks table are then projected. We can do this because we know that their IfOutlink/IfInlink value == “one.” Joins are then made on the indexes of both tables and titles.Title is projected.



```
== Optimized Logical Plan ==
Aggregate [Title#2], [Title#2]
 Project [Title#2]
  Join Inner, Some((Index#3 = Index#5))
   Project [Index#5]
    Filter (IfInlink#6 = one)
     LogicalRDD [Index#5,IfInlink#6], MapPartitionsRDD[41] at createDataFrame at Wiki_sql.scala:106
   LogicalRDD [Title#2,Index#3], MapPartitionsRDD[14] at createDataFrame at Wiki_sql.scala:76
```



```
== Optimized Logical Plan ==
Aggregate [Title#2], [Title#2]
 Project [Title#2]
  Join Inner, Some((Index#3 = Index#0))
   Project [Index#0]
    Filter (IfOutlink#1 = one)
     LogicalRDD [Index#0,IfOutlink#1], MapPartitionsRDD[11] at createDataFrame at Wiki_sql.scala:57
   LogicalRDD [Title#2,Index#3], MapPartitionsRDD[14] at createDataFrame at Wiki_sql.scala:76
```

In a similar fashion to the physical plans, the logical plans are identical except for the tables (no_outlinks and no_inlinks) used to carryout the query. My alternate logical plan is as follows:

```
== Optimized Logical Plan ==
Aggregate [Title#2], [Title#2]
 Project [Title#2]
Filter (IfInlink#6 = one)
  Join Inner, Some((Index#3 = Index#5))
   Project [Index#5,IfInlink#6]
     LogicalRDD [Index#5,IfInlink#6], MapPartitionsRDD[41] at
createDataFrame at Wiki_sql.scala:106
   LogicalRDD [Title#2,Index#3], MapPartitionsRDD[14] at
createDataFrame at Wiki_sql.scala:76
```

I pushed the filter up in the logical plan. Because of this, a majority of pages are joined with it's corresponding index if it exists in the no_outlinks/no_inlinks tables. As a result a large percentage, instead of a very small percentage, of pages are joined on their indexes. Although the number of pages with no outlinks (10,438) is significantly smaller than the number with no inlinks (1,942,943), when you compare these numbers to the total number of pages on the site (5,716,808), being able to eliminate ~80% or ~99% of all pages as early as possible in the operation can be crucial in having a plan that has low costs. Therefore, the plan that Spark constructed is certainly better than mine.

```
== Physical Plan ==
Filter (IfInlink#6 = one)
Aggregate false, [Title#2], [Title#2]
 Exchange (HashPartitioning 200)
  Aggregate true, [Title#2], [Title#2]
   Project [Title#2]
    ShuffledHashJoin [Index#5], [Index#3], BuildRight
     Exchange (HashPartitioning 200)
      PhysicalRDD [Index#5,IfInlink#6], MapPartitionsRDD[41] at
createDataFrame at Wiki_sql.scala:106
     Exchange (HashPartitioning 200)
      PhysicalRDD [Title#2,Index#3], MapPartitionsRDD[14] at
createDataFrame at Wiki_sql.scala:76
```

In this alternate plan, I eliminated a project of the index in no_inlinks. I had to eliminate this because I moved the filter that kept all elements that conform to the predicate "IfInlink#6 = one" to the very last action of the plan. Because of the elimination of the index project and the movement of the location of the IfInlink filter, unnecessary data was shuffled throughout the entirety of the plan. I could not project the index of no_inlinks without checking all of the entries that had IfInlink == one, because then the query would not execute properly. I would certainly use the physical plan generated by Spark (shown above) to execute this query rather than mine. It eliminates unnecessary data more quickly, allowing for a lower overall cost.