Problem Set 2 - Documentation

Silvey, Liam, Dhabia

Due: 20th of April, 2022

Table of Contents

Time Taken

April 14: 3*1 = 3

April 15: 3*2 = 6

April 16: 2*2 + 2 = 6

April 21: 2*4 + 2*3 = 14

April 22: 2* + 2* =

Checklist

- [x] ~~Query Plans - 6~~
  - [x] ~~A - 1~~
  - [x] ~~B - 1~~
  - [x] ~~C - 1~~
  - [x] ~~D - 1~~
  - [x] ~~E - 1~~
  - [x] ~~F - 1~~

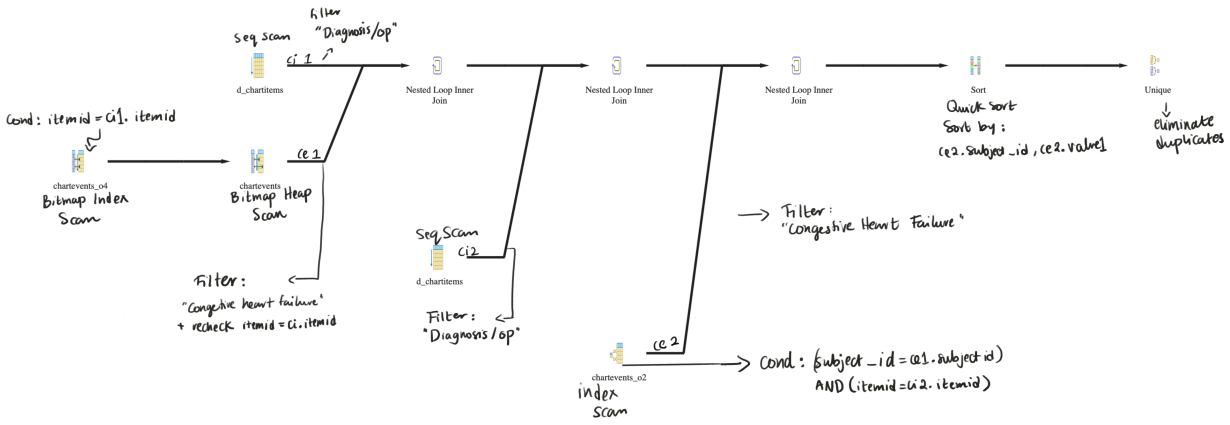
- [x] ~~Cost of Query Plans - 13~~
  - [x] ~~A - 3~~
  - [x] ~~B - 2~~
  - [x] ~~C - 2~~
  - [x] ~~D - 2~~
  - [x] ~~E - 2~~
  - [x] ~~F - 2~~
- [ ] Access Methods - 8
  - [x] ~~A - 1~~
  - [x] ~~B - 1~~
  - [x] ~~C - 1~~
  - [x] ~~D - 1~~
  - [x] ~~E - 1~~
  - [ ] F - 1
  - [ ] G - 1
  - [x] ~~H - 1~~

1.1 Query Plans

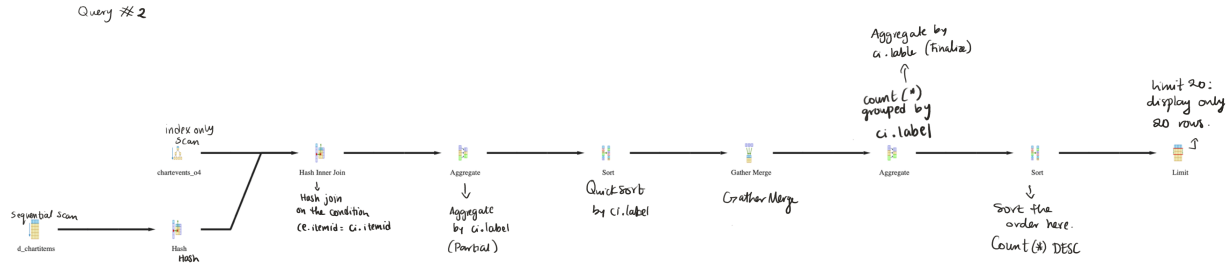
A

What physical plan does PostgreSQL use? Your answer should consist of a drawing of a query tree annotated with the access method types and join algorithms. [1 point]

Query #1



Query #2



B

Why do you think PostgreSQL selected this particular plan? [1 point]

The plans selected are cost efficient as postgres' optimizer is cost-based.

chartevents size: 34240621

d_chartitems: 4832

Query 1:

- For chartevents, it starts by performing an index scan then heap scan, this is because chartevents has several indices are built on it. For chartitems however, a seqscan is sufficient since we have only one index (the relation's primary key).

- Then the plan performed the first nested loop inner join using the relation d_chartitems as the outer relation in the join and chartevents as the inner relation in the loop.
- Picks NLJ when it's a lower amount of rows, hashing would not make sense here

Query 2:

- First, a Seq Scan is performed on d_chartitems, scanning over the entire table.
- A hashjoin is used here since it is faster to match using a hashtable than it would be doing a NLJ and scanning the relations repeatedly. The smaller relation is always the one hashed and therefore is the inner relation.
- It picks hash when there's a bigger number of predicted rows so the extra work of making a hash is worth it.

C

What does PostgreSQL estimate the size of the result set to be? [1 point]

First query: 6

Second query: 20

D

When you actually run the query, how big is the result set? [1 point]

- Number of Rows

First query: 138

Second query: 20

E

Run some queries to compute the sizes of the intermediate results in the query. Where do Post-greSQL's estimates differ from the actual intermediate result cardinalities? [1 point]

Query 1:

Size of the below intermediate result: 246, which already differs from 1. So in the first join—-the nested loops join of ce1 and ci1, the estimates already differ from the actual

result.

Query 1 intermediate query plan:

Query Editor

```
1  explain (format json)
2  SELECT DISTINCT *
3  FROM chartevents AS ce1, d_chartitems AS ci1
4  WHERE ce1.itemid=ci1.itemid
5  AND ci1.label='Diagnosis/op'
6  AND ce1.value1='CONGESTIVE HEART FAILURE';
```

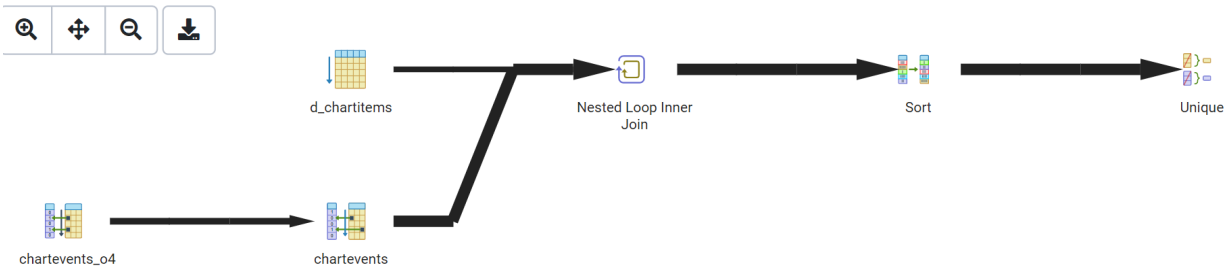
Notifications

Recorded time

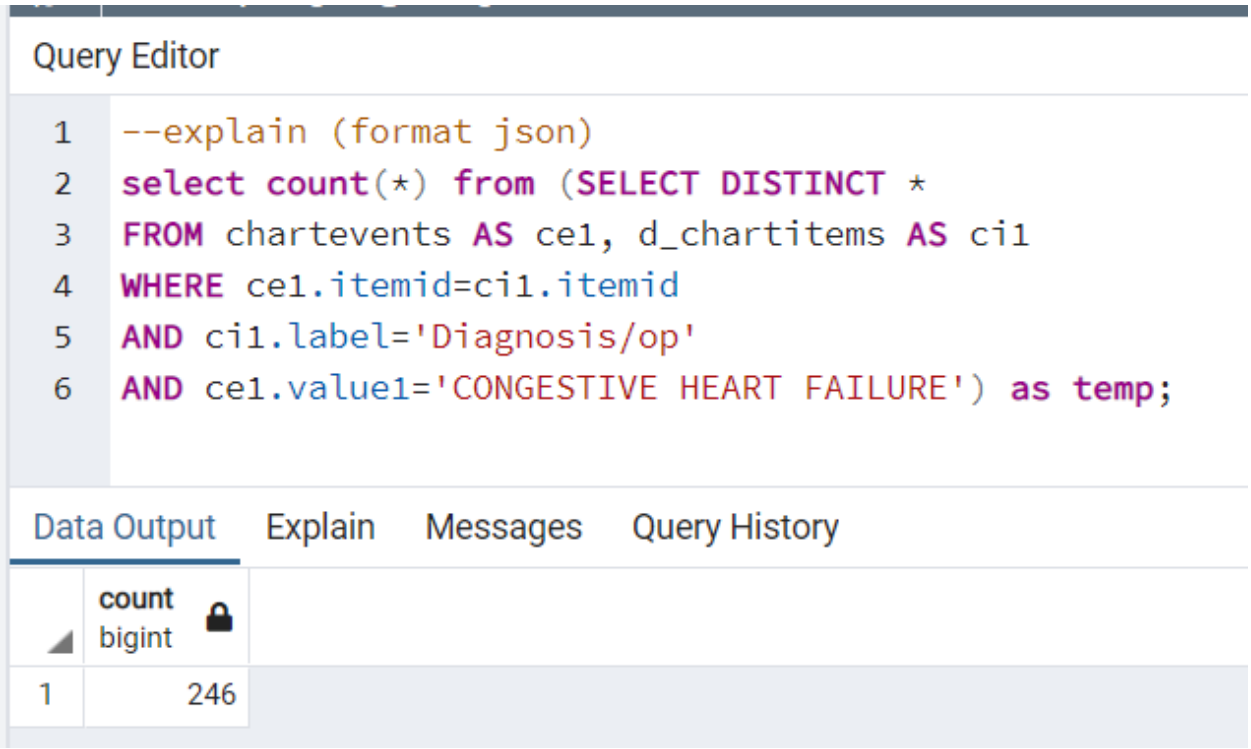
Data Output | Explain | Messages | Query History

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Unique (cost=129630.64..129630.69 rows=1 width=202)	1
2.	→ Sort (cost=129630.64..129630.65 rows=1 width=202)	1
3.	→ Nested Loop Inner Join (cost=471.47..129630.63 rows=1 width=202)	1
4.	→ Seq Scan on d_chartitems as ci1 (cost=0..90.4 rows=1 width=112) **Filter**: ((label)::text = 'Diagnosis/op'::text)	1
5.	→ Bitmap Heap Scan on chartevents as ce1 (cost=471.47..129540.15 rows=8 width=90) **Filter**: ((value1)::text = 'CONGESTIVE HEART FAILURE'::text) **Recheck Cond**: (itemid = ci1.itemid)	8
6.	→ Bitmap Index Scan using chartevents_o4 (cost=0..471.46 rows=43070 width=0) **Index Cond**: (itemid = ci1.itemid)	43070



Query 1 intermediate result:



Estimate: 1

Actual: 246

Query 2:

Although the output and the estimate are both 20, that's because we limited it to 20.

In the below screenshots, in the actual output size and the query plan estimated size differ in 2 intermediate results

Query 2 intermediate query plan estimate 1:

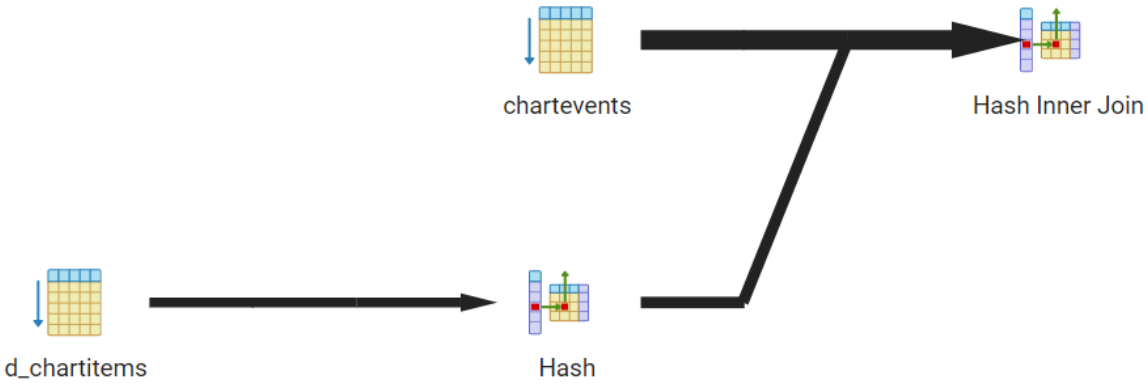
before "count" and "group by"

Query Editor

```
1  explain (format json)
2  SELECT *
3  FROM d_chartitems AS ci, chartevents AS ce
4  WHERE ci.itemid=ce.itemid;
```

Data Output | Explain | Messages | Query History

Graphical | Analysis | Statistics



Query Editor

```
1  explain (format json)
2  SELECT *
3  FROM d_chartitems AS ci, chartevents AS ce
4  WHERE ci.itemid=ce.itemid;
```

Notifications

Recorded time

Data Output | Explain | Messages | Query History

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Hash Inner Join (cost=138.72..907858.38 rows=34240620 width=202) **Hash Cond**: (ce.itemid = ci.itemid)	34240620
2.	→ Seq Scan on chartevents as ce (cost=0..817759.2 rows=34240620 width=90)	34240620
3.	→ Hash (cost=78.32..78.32 rows=4832 width=112)	4832
4.	→ Seq Scan on d_chartitems as ci (cost=0..78.32 rows=4832 width=112)	4832

Query 2 intermediate result 1:

Query Editor

```
1  --explain (format json)
2  select count(*) from (SELECT *
3  FROM d_chartitems AS ci, chartevents AS ce
4  WHERE ci.itemid=ce.itemid) as temp;
```

Data Output | Explain | Messages | Query History

	count bigint
1	34240621

Estimate: 34240620

Actual: 34240621

Query 2 intermediate query plan estimate 2:

Right after “group by”

Query Editor

```
1  explain (format json)
2  SELECT ci.label, count(*)
3  FROM d_chartitems AS ci, chartevents AS ce
4  WHERE ci.itemid=ce.itemid group by ci.label;
```

Notifications

Recorded time

Data Output | Explain | Messages | Query History

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Aggregate (cost=540225.69..541449.87 rows=4832 width=21)	4832
2.	→ Gather Merge (cost=540225.69..541353.23 rows=9664 width=21)	9664
3.	→ Sort (cost=539225.67..539237.75 rows=4832 width=21)	4832
4.	→ Aggregate (cost=538881.67..538929.99 rows=4832 width=21)	4832
5.	→ Hash Inner Join (cost=139.16..467547.04 rows=14266925 width=13) Hash Cond: (ce.itemid = ci.itemid)	14266925
6.	→ Index Only Scan using chartevents_o4 on chartevents as ce (cost=0.44..429924.79 rows=1426692...	14266925
7.	→ Hash (cost=78.32..78.32 rows=4832 width=17)	4832
8.	→ Seq Scan on d_chartitems as ci (cost=0..78.32 rows=4832 width=17)	4832

Query 2 intermediate result 2:

Query Editor

```
1  --explain (format json)
2  select count(*) from (SELECT ci.label, count(*)
3  FROM d_chartitems AS ci, chartevents AS ce
4  WHERE ci.itemid=ce.itemid group by ci.label) as temp;
```

Data Output | Explain | Messages | Query History

	count bigint
1	2136

Estimate: 4832

Actual: 2136

(Just for reference) Query 2 original estimate:

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Limit (cost=541578.45..541578.5 rows=20 width=21)	20
2.	→ Sort (cost=541578.45..541590.53 rows=4832 width=21)	4832
3.	→ Aggregate (cost=540225.69..541449.87 rows=4832 width=21)	4832
4.	→ Gather Merge (cost=540225.69..541353.23 rows=9664 width=21)	9664
5.	→ Sort (cost=539225.67..539237.75 rows=4832 width=21)	4832
6.	→ Aggregate (cost=538881.67..538929.99 rows=4832 width=21)	4832
7.	→ Hash Inner Join (cost=139.16..467547.04 rows=14266925 width=13) **Hash Cond**: (ce.itemid = ci.itemid)	14266925
8.	→ Index Only Scan using chartevents_o4 on chartevents as ce (cost=0.44..42...	14266925
9.	→ Hash (cost=78.32..78.32 rows=4832 width=17)	4832
10.	→ Seq Scan on d_chartitems as ci (cost=0..78.32 rows=4832 width=17)	4832

F

Given the tables and indices we have created, do you think PostgreSQL selected the best plan? You can list all indices with \di, or list the indices for a particular table with \d tablename. If not, what would be a better plan? Why? [1 point]

PostgreSQL not always selects the best plan, but what it does is good enough and quite similar to the best plan.

Indices for Query 1 and Query 2 tables:

chartevents:

```
Indexes:
    "chartevents_o1" btree (cuid)
    "chartevents_o2" btree (subject_id, itemid)
    "chartevents_o3" btree (subject_id)
    "chartevents_o4" btree (itemid)
    "chartevents_o5" btree (icustay_id)
    "chartevents_o6" btree (charttime)
    "chartevents_o7" btree (cgid)
```

d_chartitems:

In Query 1, index scans were performed on indices chartevents_o4 and chartevents_o2, and sequential scans were performed on d_chartitems twice, tasks that the primary key index surely helped with performance-wise. Indexing them all helps with the joining of multiple tables on different columns.

In Query 2, only one index-only scan was performed on index chartevents_o4. It joins only once and the index is enough for the query optimization.

Also, judging from the visualizations produced by Postgres, the indexing, join, and join algorithms are good enough. They are cost-efficient, as Postgres' optimizer is cost-based.

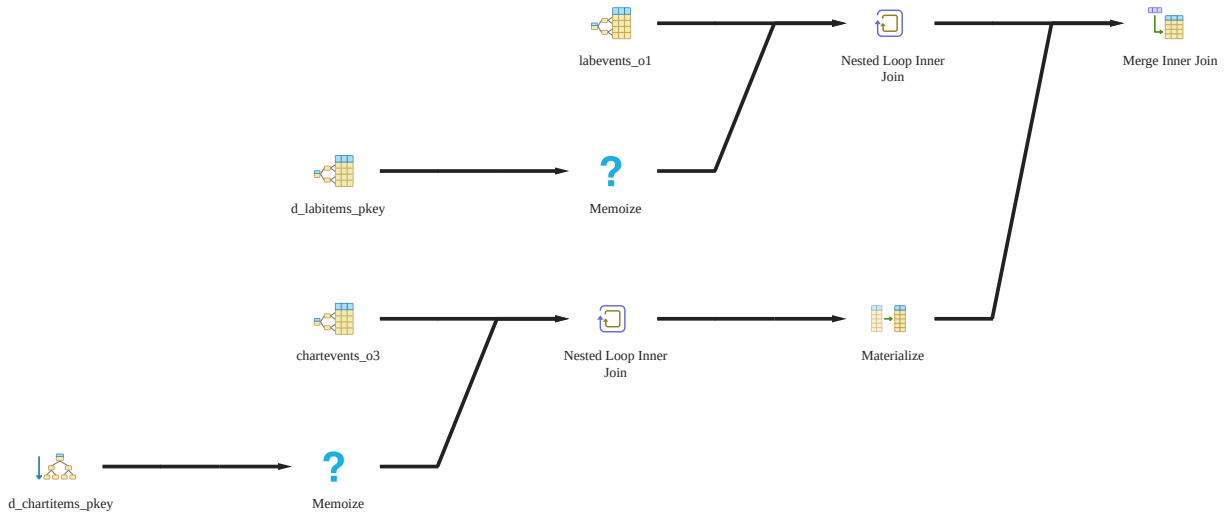
2.0 Estimating the Cost of Query Plans

A

Notice that the query plans for the two queries above are different, even though they have the same general form. Explain the difference in the query plan that PostgreSQL chooses, and explain why you think the plans are different. [3 points]

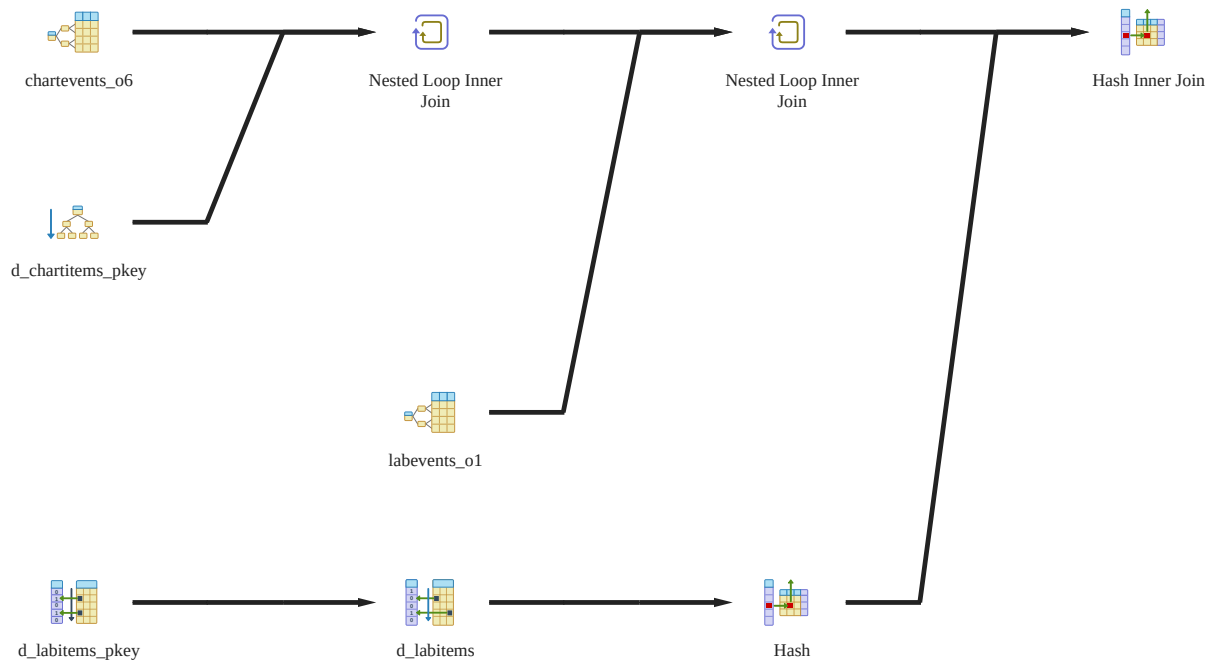
```
explain (format json)
SELECT l.test_name
FROM d_chartitems AS c, chartevents AS ce,
labevents AS le, d_labitems AS l
WHERE c.itemid=ce.itemid
AND ce.subject_id=le.subject_id
AND le.itemid=l.itemid
AND ce.charttime > '3000-02-17 00:00:00'
AND l.itemid>50700;
```

Query 1 query plan:



	#	Node	Rows	
			Actual	Loops
	1.	→ Merge Inner Join (rows=166735 loops=1)	166735	1
	2.	→ Nested Loop Inner Join (rows=21 loops=1)	21	1
	3.	→ Index Scan using labevents_o1 on labevents as le (rows=3740682 loops=1)	3740682	1
	4.	→ Memoize (rows=0 loops=3740682) **Buckets**: **Batches**: **Memory Usage**: 44 kB	0	3740682
	5.	→ Index Scan using d_labitems_pkey on d_labitems as l (rows=0 loops=652) **Index Cond**: ((itemid = le.itemid) AND (itemid > 50700))	0	652
	6.	→ Materialize (rows=14912624 loops=1)	14912624	1
	7.	→ Nested Loop Inner Join (rows=14873632 loops=1)	14873632	1
	8.	→ Index Scan using chartevents_o3 on chartevents as ce (rows=14873632 loops=1) **Filter**: (charttime > '3000-02-17 00:00:00'::timestamp without time zone) **Rows Removed by Filter**: 13878870	14873632	1
	9.	→ Memoize (rows=1 loops=14873632) **Buckets**: **Batches**: **Memory Usage**: 171 kB	1	14873632
	10.	→ Index Only Scan using d_chartitems_pkey on d_chartitems as c (rows=1 loops=16… **Index Cond**: (itemid = ce.itemid)	1	1680

Query 2 query plan:



Explain

Graphical | Analysis | Statistics

#	Node	Rows Actual	Loops
1.	→ Hash Inner Join (rows=0 loops=1) **Hash Cond**: (le.itemid = l.itemid)	0	1
2.	→ Nested Loop Inner Join (rows=0 loops=1)	0	1
3.	→ Nested Loop Inner Join (rows=0 loops=1)	0	1
4.	→ Index Scan using chartevents_o6 on chartevents as ce (rows=0 loops=1) **Index Cond**: (charttime > '3600-02-17 00:00:00'::timestamp without time zone)	0	1
5.	→ Index Only Scan using d_chartitems_pkey on d_chartitems as c (rows=0 loops=0) **Index Cond**: (itemid = ce.itemid)	0	0
6.	→ Index Scan using labevents_o1 on labevents as le (rows=0 loops=0) **Index Cond**: (subject_id = ce.subject_id)	0	0
7.	→ Hash (rows=0 loops=0)	0	0
8.	→ Bitmap Heap Scan on d_labitems as l (rows=0 loops=0) **Recheck Cond**: (itemid > 50700) **Heap Blocks**: exact=0	0	0
9.	→ Bitmap Index Scan using d_labitems_pkey (rows=0 loops=0) **Index Cond**: (itemid > 50700)	0	0

Despite the only difference between Query 1 and Query 2 being the ce.charttime specified (**3000**-02-17 vs **3600**-02-17). As a result, there is a slight difference in the range of selection, and so the number of rows being selected will be a bit different.

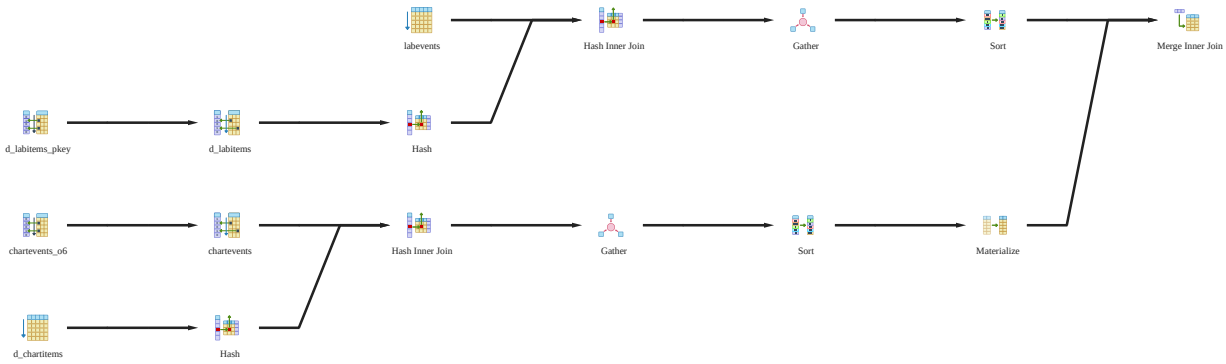
Postgres will get an intermediate result with more rows in the first query than in the second one.

For Query 1, Postgres chose to memoize. It has a much larger intermediate result and would be expensive to read the original table every time from disk. Query 2 has a much smaller (zero) intermediate result from the selection (row = 0) and it makes sense to not materialize.

For Query 2, Postgres chose to follow a left-deep tree, but left-deep plan does not suit query 1 so well because query 1 has significantly larger intermediate results than query 2, and following this “default” left-deep plan will lead to worse cost. Postgres avoided it.

B

What is PostgreSQL doing for this query? How is it different from the previous two plans that are generated? You may find it useful to draw out the plans (or use pgadmin3) to get a visual representation of the differences, though you are not required to submit drawings of the plans in your answer. [2 points]



Explain

Graphical | Analysis | Statistics

	#	Node	Rows Actual	Loops
	1.	→ Merge Inner Join (rows=0 loops=1)	0	1
	2.	→ Sort (rows=21 loops=1)	21	1
	3.	→ Gather (rows=21 loops=1)	21	1
	4.	→ Hash Inner Join (rows=7 loops=3) **Hash Cond**: (le.itemid = l.itemid)	7	3
	5.	→ Seq Scan on labevents as le (rows=1246894 loops=3)	1246894	3
	6.	→ Hash (rows=34 loops=3) **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 10 kB	34	3
	7.	→ Bitmap Heap Scan on d_labitems as l (rows=34 loops=3) **Recheck Cond**: (itemid > 50700) **Heap Blocks**: exact=1	34	3
	8.	→ Bitmap Index Scan using d_labitems_pkey (rows=34 loops=3) **Index Cond**: (itemid > 50700)	34	3
	9.	→ Materialize (rows=94099 loops=1)	94099	1
	10.	→ Sort (rows=94099 loops=1)	94099	1
	11.	→ Gather (rows=126926 loops=1)	126926	1
	12.	→ Hash Inner Join (rows=42309 loops=3) **Hash Cond**: (ce.itemid = c.itemid)	42309	3
	13.	→ Bitmap Heap Scan on chartevents as ce (rows=42309 loops=3) **Recheck Cond**: (charttime > '3500-02-17 00:00:00'::timestamp without time zone) **Heap Blocks**: exact=641	42309	3
	14.	→ Bitmap Index Scan using chartevents_o6 (rows=126926 loops=1) **Index Cond**: (charttime > '3500-02-17 00:00:00'::timestamp without time zone)	126926	1
	15.	→ Hash (rows=4832 loops=3) **Buckets**: 8192 **Batches**: 1 **Memory Usage**: 234 kB	4832	3
●	16.	→ Seq Scan on d_chartitems as c (rows=4832 loops=3)	4832	3

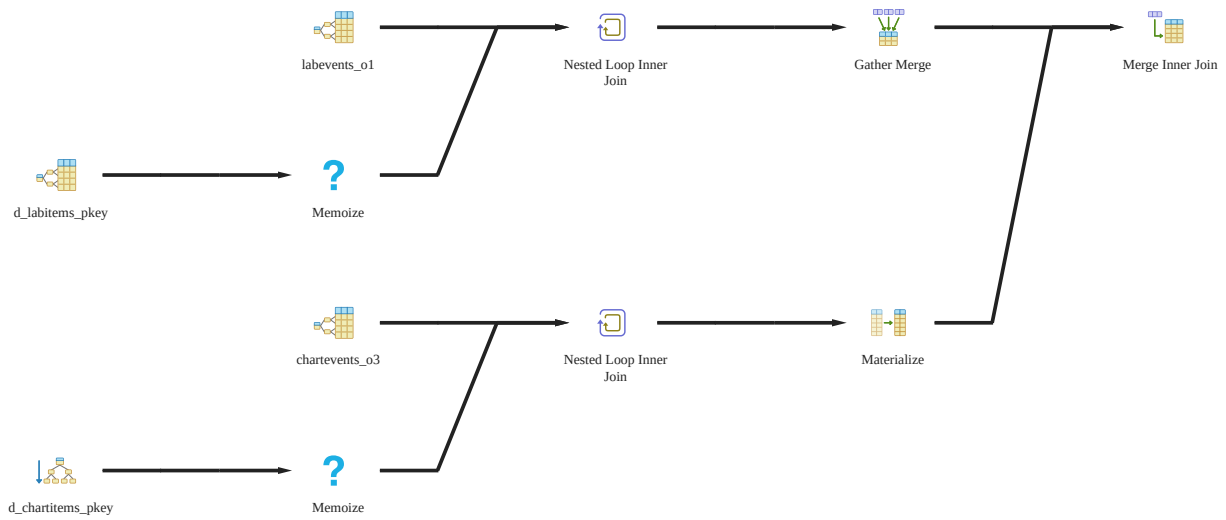
Let's call this query **Query 3.** For this query the selection was '>3500-02-17', it's different from the previous 2 plans because the selection range is greater than that of query 2, but smaller than that of query 1.

One main difference is that this query plan is much longer than the previous two query plans, as it involves two hash inner joins and a final merge inner join, whereas the previous two query plans primarily rely upon nested loop inner joins and then a final merge inner join and hash inner join, respectively.

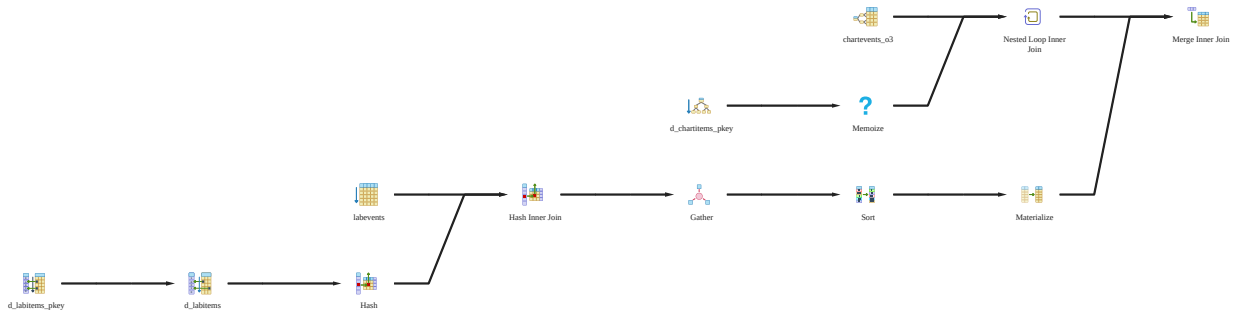
C

Run some more EXPLAIN commands using this query, sweeping the constant after the '>' sign from 3000 to 3600. For what value of charttime does PostgreSQL switch plans? Note: There might be additional plans other than these three. If you find this to be the case, please write the estimated last value of charttime for which PostgreSQL switches query plans. [2 points]

For charttime values from 3000 to 3406, PostgreSQL uses the plan of Query 1. This all changes at value **3407,** where PostgreSQL performs an additional ‘gather merge’ on the left (topmost) branch:

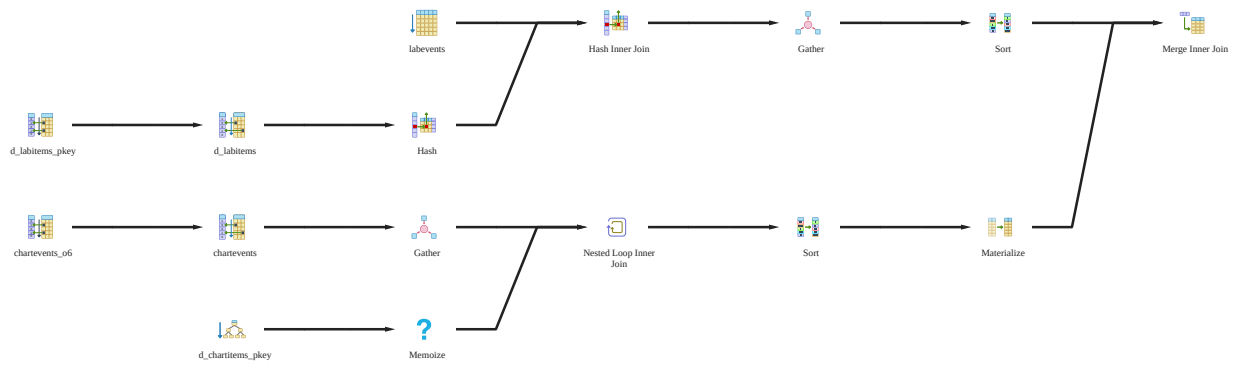


This above query plan is utilized from 3407 to 3426. However, at value **3427**, PostgreSQL decides to switch to the following plan:



#	Node	Actual	Loops
1.	→ Merge Inner Join (rows=3560 loops=1)	3560	1
2.	→ Nested Loop Inner Join (rows=2115469 loops=1)	2115469	1
3.	→ Index Scan using chartevents_o3 on chartevents as ce (rows=2... **Filter**: (charttime > '3427-02-17 00:00:00'::timestamp without time zone) **Rows Removed by Filter**: 26659458	2115469	1
4.	→ Memoize (rows=1 loops=2115469) **Buckets**: **Batches**: **Memory Usage**: 108 kB	1	2115469
5.	→ Index Only Scan using d_chartitems_pkey on d_chartitems... **Index Cond**: (itemid = ce.itemid)	1	1063
6.	→ Materialize (rows=3579 loops=1)	3579	1
7.	→ Sort (rows=21 loops=1)	21	1
8.	→ Gather (rows=21 loops=1)	21	1
9.	→ Hash Inner Join (rows=7 loops=3) **Hash Cond**: (le.itemid = l.itemid)	7	3
10.	→ Seq Scan on labevents as le (rows=1246894 loo...	1246894	3
11.	→ Hash (rows=34 loops=3)	34	3
12.	→ Bitmap Heap Scan on d_labitems as l (rows... **Recheck Cond**: (itemid > 50700) **Heap Blocks**: exact=1	34	3
13.	→ Bitmap Index Scan using d_labitems_p... **Index Cond**: (itemid > 50700)	34	3

Then, at value **3428**, PostgreSQL switches again to the plan of Query 3. This is sustained through value **3520**, and at value **3521** the following plan is used:



PostgreSQL frequently switches plans in this period, however the last value of charttime for which PostgreSQL switches query plans is **3586**, where the plan of Query 2 is used.

In summary, the breakpoints for charttime values from 3000 to 3600 are:

3000 (Q1) — 3407 — 3427 — 3428 (Q3) — 3521 — frequent switches — 3586 (Q2)

D

Why do you think it decides to switch plans at the values you found? Please be as quantitative as possible in your explanation. [2 points]

I will focus on the important switches at 3428 to Query 3 and 3586 to Query 2:

Switch 1 (3427 → 3428): According to EXPLAIN, PostgreSQL expects 2115469 rows to be returned from index scanning chartevents with charttime > 3427. With charttime > 3428, however, PostgreSQL expects 808831 rows to be returned from a sequential scan of chartevents. First of all, the substantial difference in expected rows likely explains the change in query plans and use of a nested loop inner join for values > 3427 and an additional hash inner join used for values > 3428.

Switch 2 (3585 → 3586): For chartttime > 3585, the bitmap index scan performed by PostgreSQL expects 285 rows to be returned. However, for charttime values > 3586, an index scan is performed which expects 0 returned rows. This drastic difference in number of expected rows explains the shift in query plans and the use of a much simpler, shorter query plan for values > 3586.

E

Suppose the crossover point (from the previous question) between queries 2 and 3 is charttime23. Compare the actual running times corresponding to the two alternative query plans at the crossover point. How much do they differ? Do the same for all switch points. Inside psql, you can measure the query time by using the \timing command to turn timing on for all queries. To get accurate timing, you may also wish to redirect output to /dev/null, using the command \o /dev/null; later on you can stop redirection just by issuing the \o command without any file name. You may also wish to run each query several times, throwing out the first time (which may be longer if the data is not resident in the buffer pool) and averaging the remaining runs. [2 points]

Again, I will focus on the important switch points, the switch to Query 3 and switch to Query 2. Each query was run 3 times and the average value was calculated.

Switch 1 (3427 → 3428): Before the crossover/switch point, running the query resulted in an average runtime of 4848.9 ms. After the switch point, the query took an average of 2502.7 ms.

Switch 2 (3585 → 3586): Before the switch, the query takes an average of 174.4 ms. Afterwards, the query takes an average of 0.992 ms.

F

Based on your answers to the previous two problems, are those switch points actually the best place to switch plans, or are they overestimate/underestimate of the best crossover points? If they are not the actual crossover point, can you estimate the actual best crossover points without running queries against the database? State assumptions underlying your answer, if any. [2 points]

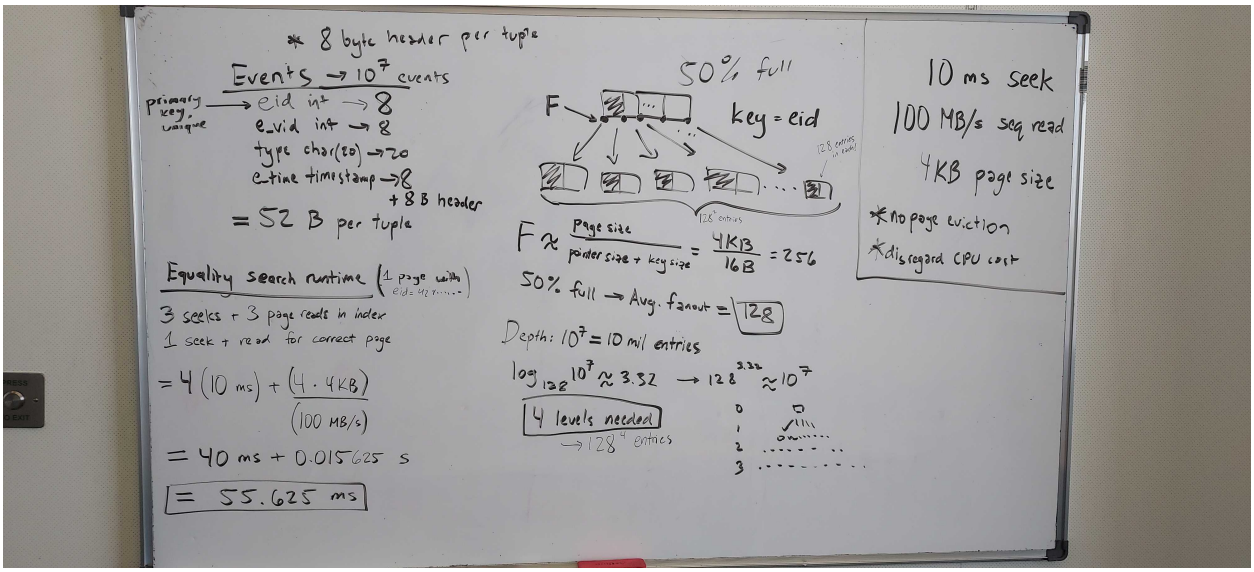
Based on the runtimes calculated in part e, both switch points seems like good, reasonable places to switch. Both cases have much faster runtimes after the switch than beforehand, and, drawing upon part d, this is likely due to a substantial difference in the number of expected rows to be returned from each query before/after the switch. Therefore, it appears that Postgres did a good job as both crossover points seem to be good places to switch.

3.0 Access Methods

A

Suppose there is a B+Tree on events.eid. Considering all possible plans the executor might use,
estimate the runtime of the fastest plan to evaluate this query when the index is unclustered.

[1 point]



Estimated runtime of the fastest plan given an unclustered index is 55.625 ≈ 56 ms.

B

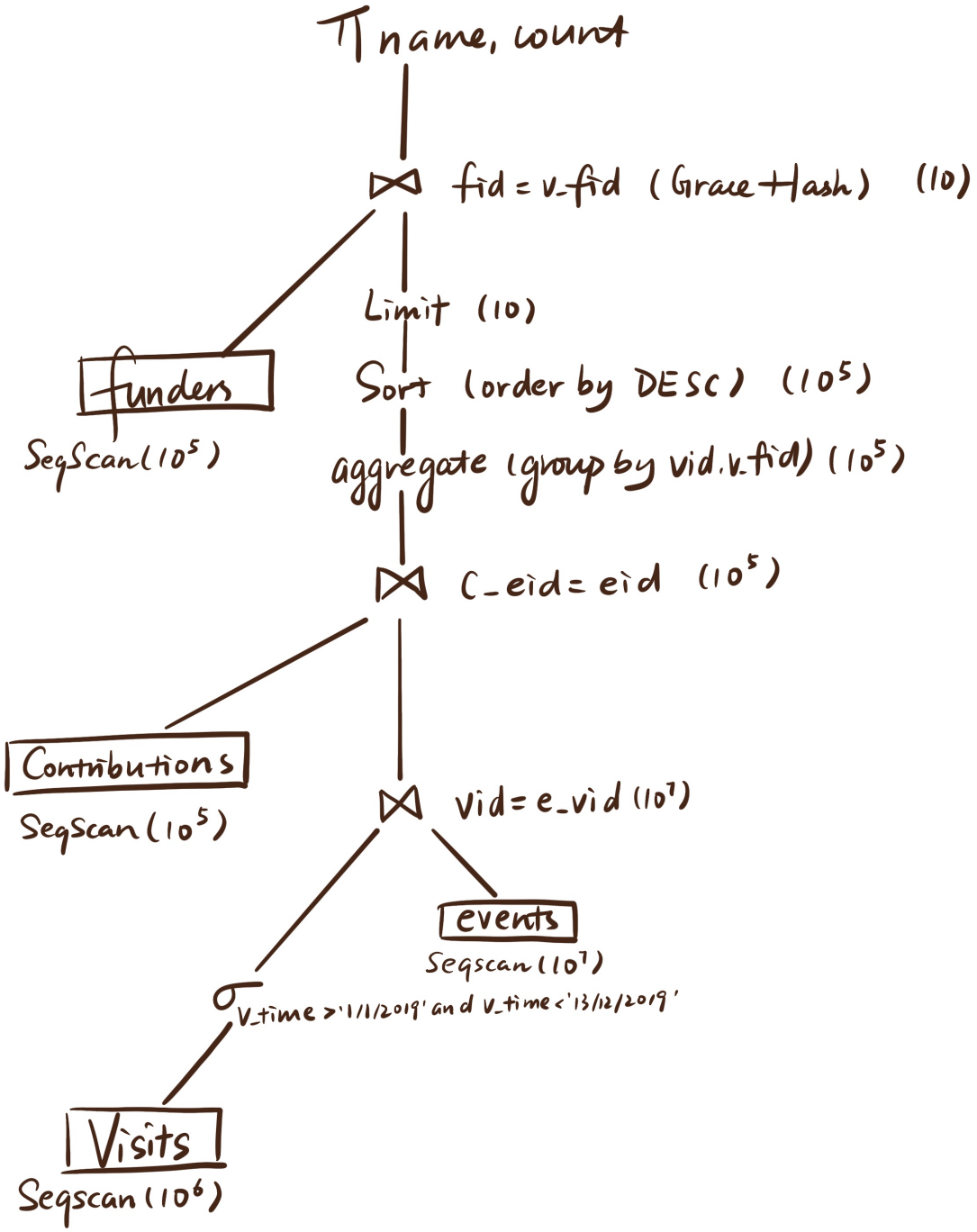
Consider the same query and the same B+Tree on events.eid. Considering all possible plans the executor might use, estimate the runtime of the fastest plan to evaluate this query when the index is clustered. [1 point]

As this query is an equality search for an event with a specific eid (primary key) = 548291132, only 1 row should be returned, if any. Therefore a clustered index would not impact the runtime of this query as clustered indexes outperform unclustered indexes with ranged searches, not equality searches. The estimated runtime should remain the same: 55.625 ≈ 56 ms.

C

Suppose only heap files are available (i.e., there are no indexes), and that the system has both grace (hybrid) hash and merge join methods available to it. Draw the query plan you think the database system would use to evaluate this query. [1 point]

Since we do not have any information regarding how much memory the system has, it seems prudent to stick with grace hash joins to ensure the outer relation’s hash table will fit in memory, partitioning large tables as needed. The drawing is as follows:


π name, count
fid = v_fid (Grace Hash) (10)
Limit (10)
Sort (order by DESC) (10^5)
aggregate (group by vid, v_fid) (10^5)
C_eid = eid (10^5)
vid = e_vid (10^7)
σ v_time > '1/1/2019' and v_time < '13/12/2019'
funders
SeqScan(10^5)
Contributions
Seqscan(10^5)
events
Seqscan(10^7)
Visits
Seqscan(10^6)

D

For each node in your query plan indicate (on the drawing, if you wish), the approximate output cardinality (number of tuples produced.) [1 point]

Included in the drawing above.

E

Estimate the runtime of the plan you drew, in seconds. [1 point]

Table name	# pages	table size
Funders	10^5 (116B) / 4KB = 2,900	11.6MB
Visits	10^6 (40B) / 4KB = 10,000	40MB
Events	10^7 (52B) / 4KB = 130,000	520MB
Contributions	10^5 (32B) / 4KB = 800	3.2MB

Scan of Visits table: (10^6 * 40) / (100*1000000) = **0.4 seconds**

After selection of v_time: Visit table size now reduces to 0.5 * 20^6 * 40B = 2*10^7 B

Scan of Events table: (10^7 * 52) / (100*1000000) = **5.2 seconds**

Join between Visits and Events tables: not counted as I/O, produces a new table of size (0.5*10^6) * (40 + 52) = 4.6*10^7 B

Scan of Contributions table: (10^5 * 32) / (100*1000000) = **0.032 seconds**

Join between Contributions and the intermediate result: not counted as I/O, produces a new table of size (0.5*10^6) * (40 + 52 + 32) = 6.2*10^7 B

Aggregation, sort, and limit 10: not counted as I/O

Scan of Funders table: (10^5 * 116) / (100*1000000) = **0.116 seconds**

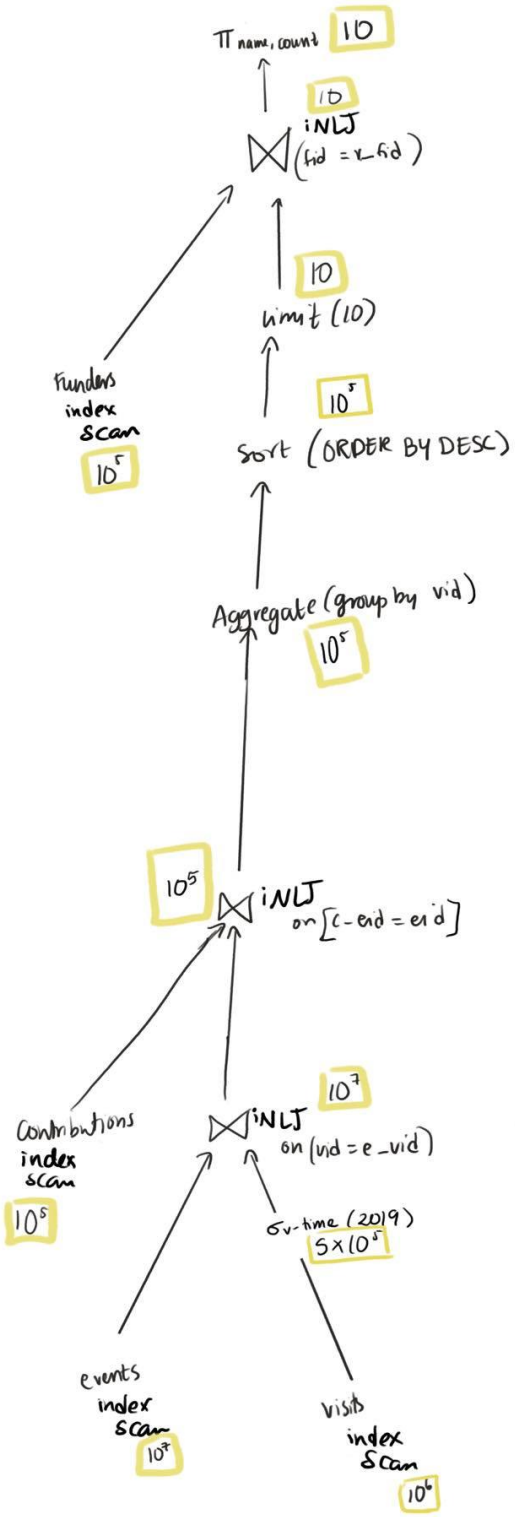
Join of Funders and the intermediate result: + selection: not counted as I/O

Write the final result: 10 tuples each 100+8+8 = 116 B, (10 * 116) / (100*1000000) = **0.0000116 seconds**

Summing up all the above: 0.4 + 5.2 + 0.032 + 0.116 + 0.0000116 = **5.7480116 seconds**

F

Now, suppose that there are clustered B+Trees on fid, vid, eid, and cid, and an unclustered B+Tree on e_vid. Draw the new plan you think the database would use and estimate its runtime. [1 point]


π name, count
10
10
iNLJ
(fid = r_fid)
10
limit (10)
Founders
index
scan
10^5
10^5
Sort (ORDER BY DESC)
Aggregate (group by vid)
10^5
10^5
iNLJ
on [c-eid = eid]
Contributions
index
scan
10^5
10^7
iNLJ
on (vid = e_vid)
σv-time (2019)
5×10^5
events
index
scan
10^7
visits
index
scan
10^6

G

Suppose that there are clustered B+Trees on fid, v_time, e_vid and c_eid, and unclustered B+Trees on cid and vid. Draw the plan you think the database would use and estimate its runtime. [1 point]

H

Is it possible to flatten the query in the previous problem into an un-nested representation? If so,write the flattened representation. If not, state why. [1 point]

Yes, it is possible, but the new query would be a lot less cost-efficient.

We can flatten it to look something like this following mock query:

```
SELECT name, vid, v_fid, COUNT( * ) AS count
FROM visits, events, contributions, funders,
WHERE vid = e_vid
AND c_eid = eid
AND fid = v_fid
AND v_time BETWEEN '1/1/2019' and '31/12/2019'
GROUP BY vid, v_fid
ORDER BY COUNT( * ) DESC
LIMIT 10;
```

This way we will be joining very large tables. When it was nested, we only join funders with 10 tuples. Now we could be joining funders with the intermediate result produced by all other tables, which will have much more rows than 10. It would be better to do “LIMIT 10” as early as possible.