Srivathsava Rangarajan

Explain Plans and You.

2019/09/13

Disclaimer: Here there may be no dragons.

What is this talk not about?

- Innovation, development and advances in PostgreSQL and new features
- Brilliant depth first analysis of a single facet of PostgreSQL

What is this talk about?

- What is an explain plan.
- Analyzing the anatomy of the explain plan.
- Forewords on complexity, indices, joins and sequential scans wrt explain plans

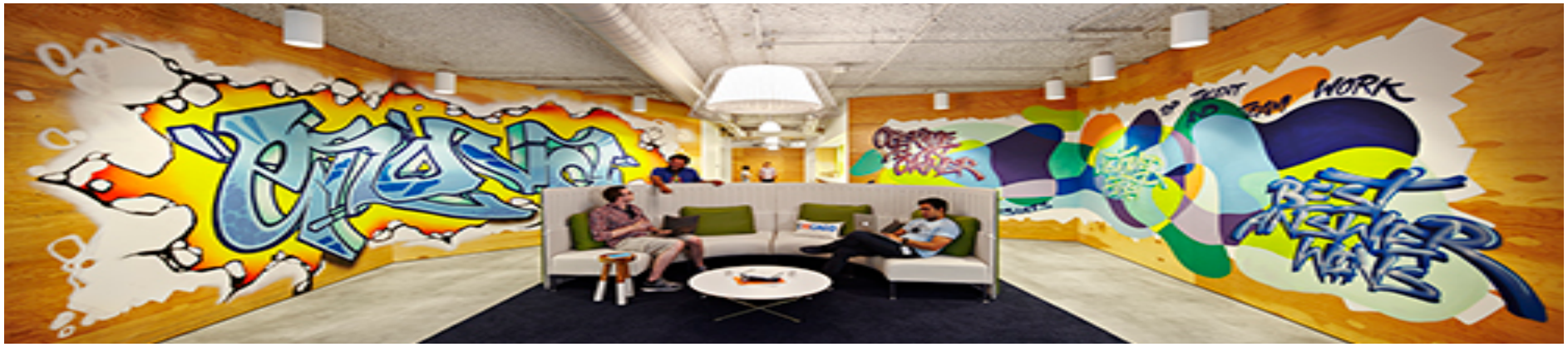
Why are you giving this talk?

- I assume that I can't be alone in having had a non-traditional introduction to RDBMSes given the low barrier to entry for SQL
- I assume that not many non-command line people may even be aware of EXPLAIN

Why are *you* giving this talk?

- I am the principal software engineer for underwriting services at a billion$ financial company
- My team owns 15+ services that use more than 6 types of persistence technologies
- My team owns services running over a couple of terabytes of transactional PostgreSQL data that need to have sub 100ms response times

Where is it you work again?



Enova: Chicago based FinTech Lending and Analytics/aa/Service

Sizeable PostgreSQL shop:

- 300+ production clusters, 500+ production databases
- > 10 databases of TB+ size

Great:

- People, value, leadership, opportunities

As is every company, we are hiring!

- If interested, please contact me: srangarajan@enova.com

A Gentle™ Introduction to:

What in the world the PostgreSQL optimizer is doing with your poor query

enova

Why and when should you care?

First, remember that PostgreSQL runs a cost based optimizer.

- In other words, PostgreSQL decides using statistics as to what the best approach to running your query is

Explain plans lie at the heart of how fast your query is going to run.

- It is not absolute, hardware plays a significant role.
- Understanding it will help you squeeze every bit you can out of the optimizer.

We can't "control" the optimizer, but if we understand why it does what it does, we can have a better relationship with it.

- Optimizer can't be hinted.
- But the optimizer is smart.

What is fast today, may turn out to be slow tomorrow.

- Explain plans may help you foresee problems that are coupled with volume.
- Prevention is better than cure.

How do I?

EXPLAIN [query]

- Explains the plan/play the optimizer *thinks* it is going to run to execute your query using estimates.

EXPLAIN ANALYZE [*query].*

- Explains the plan/play the optimizer *actually executed* by *actually executing* your query. Interesting to note that estimates may be off.

The Anatomy of an Explain Plan

```
                                                                         QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=22.67..22.70 rows=2 width=44) (actual time=0.066..0.067 rows=1 loops=1)
   ->  Sort  (cost=22.67..22.68 rows=3 width=44) (actual time=0.065..0.065 rows=2 loops=1)
         Sort Key: la.account_id, la.external_entity_id, la.created_at
         Sort Method: quicksort  Memory: 25kB
         ->  Hash Join  (cost=10.66..22.65 rows=3 width=44) (actual time=0.048..0.052 rows=2 loops=1)
               Hash Cond: (la.loan_application_status_id = loan_application_statuses.loan_application_status_id)
               ->  Bitmap Heap Scan on loan_applications la  (cost=9.21..21.16 rows=3 width=36) (actual time=0.022..0.025 rows=2 loops=1)
                     Recheck Cond: ((account_id = ANY ('{7812011}'::integer[])) OR (external_entity_id = ANY ('{NULL}'::integer[])))
                     ->  BitmapOr  (cost=9.21..9.21 rows=3 width=0) (actual time=0.016..0.016 rows=0 loops=1)
                           ->  Bitmap Index Scan on index_loan_applications_on_account_id  (cost=0.00..4.62 rows=3 width=0) (actual time=0.014..0.014 rows=2 loops=1)
                                 Index Cond: (account_id = ANY ('{7812011}'::integer[]))
                           ->  Bitmap Index Scan on index_loan_applications_on_external_entity_id  (cost=0.00..4.60 rows=1 width=0) (actual time=0.001..0.001 rows=0 loops=1)
                                 Index Cond: (external_entity_id = ANY ('{NULL}'::integer[]))
               ->  Hash  (cost=1.20..1.20 rows=20 width=16) (actual time=0.016..0.016 rows=20 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 1kB
                     ->  Seq Scan on loan_application_statuses  (cost=0.00..1.20 rows=20 width=16) (actual time=0.004..0.007 rows=20 loops=1)
 Total runtime: 0.122 ms
(17 rows)
```

Node: What is happening in this step? Feed result to parent Node.

Relation: What is it happening on? Table or result of child Node?

Cost: ***Relatively*** how expensive is this step?

Modifier: Tweak result before handoff.

Rows: How many rows will be returned by this Node.

Loops: How many times will this step be executed.

The Negative of an Explain Plan

```sql
SELECT DISTINCT ON(la.account_id, la.external_entity_id)
        ...
FROM loan_applications la
JOIN loan_application_statuses USING (loan_application_status_id)
WHERE la.account_id = ANY(ARRAY[7812011]::INTEGER[])
OR la.external_entity_id = ANY(ARRAY[NULL]::INTEGER[])
ORDER BY la.account_id, la.external_entity_id, la.created_at DESC;
```

```
                                   QUERY PLAN
----------------------------------------------------------------------------------
 Unique  (cost=22.67..22.70 rows=2 width=44) (actual time=0.066..0.067 rows=1 loops=1)
   ->  Sort  (cost=22.67..22.68 rows=3 width=44) (actual time=0.065..0.065 rows=2 loops=1)
         Sort Key: la.account_id, la.external_entity_id, la.created_at
         Sort Method: quicksort  Memory: 25kB
         ->  Hash Join  (cost=10.66..22.65 rows=3 width=44) (actual time=0.048..0.052 rows=2 loops=1)
               Hash Cond: (la.loan_application_status_id = loan_application_statuses.loan_application_status_id)
               ->  Bitmap Heap Scan on loan_applications la  (cost=9.21..21.16 rows=3 width=36) (actual time=0.022..0.025 rows=2 loops=1)
                     Recheck Cond: ((account_id = ANY ('{7812011}'::integer[])) OR (external_entity_id = ANY ('{NULL}'::integer[])))
                     ->  BitmapOr  (cost=9.21..9.21 rows=3 width=0) (actual time=0.016..0.016 rows=0 loops=1)
                           ->  Bitmap Index Scan on index_loan_applications_on_account_id  (cost=0.00..4.62 rows=3 width=0) (actual time=0.014..0.014 rows=2 loops=1)
                                 Index Cond: (account_id = ANY ('{7812011}'::integer[]))
                           ->  Bitmap Index Scan on index_loan_applications_on_external_entity_id  (cost=0.00..4.60 rows=1 width=0) (actual time=0.001..0.001 rows=0 loops=1)
                                 Index Cond: (external_entity_id = ANY ('{NULL}'::integer[]))
               ->  Hash  (cost=1.20..1.20 rows=20 width=16) (actual time=0.016..0.016 rows=20 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 1kB
                     ->  Seq Scan on loan_application_statuses  (cost=0.00..1.20 rows=20 width=16) (actual time=0.004..0.007 rows=20 loops=1)
 Total runtime: 0.122 ms
(17 rows)
```

A Word on Costs

It's all relative.

- Don't get caught up in the number. It means as much as saying I have a power level of 9001.

Computed based on a combination of I/O, CPU and memory costs.

- Weighted based on numbers set in configuration.

But, since it's relative, you can COMPARE costs between Nodes to identify/diagnose the areas for optimization.

Live Experiments: Setup

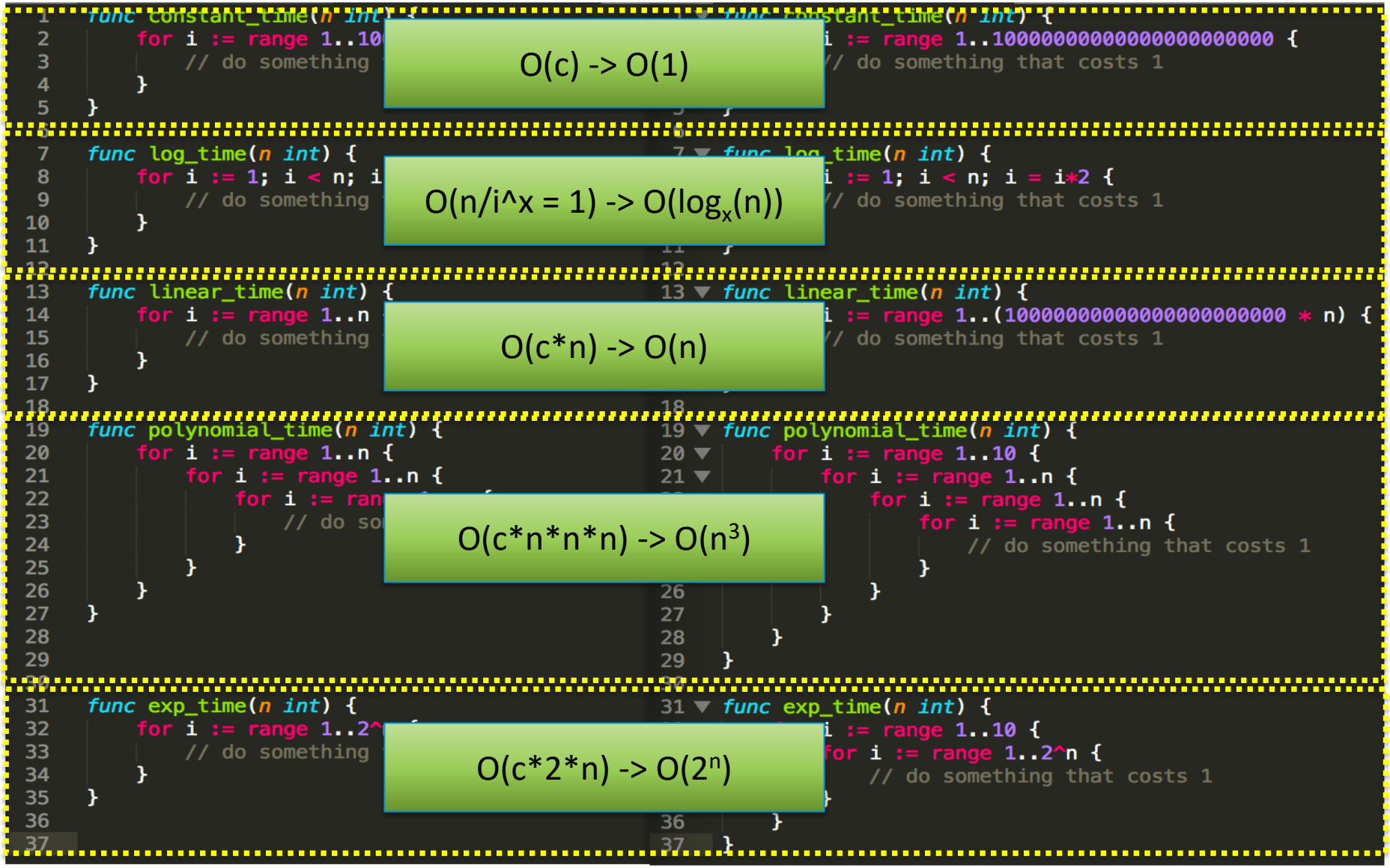
```
sandbox_development=# \d hashes
                                Table "public.hashes"
  Column   |  Type   | Collation | Nullable |                  Default
-----------+---------+-----------+----------+-------------------------------------------
 hash_id   | integer |           | not null | nextval('hashes_hash_id_seq'::regclass)
 algorithm | text    |           |          |
Indexes:
    "hashes_pkey" PRIMARY KEY, btree (hash_id)
Referenced by:
    TABLE "hash_test" CONSTRAINT "hash_test_hash_id_fkey" FOREIGN KEY (hash_id) REFERENCES hashes(hash_id)

sandbox_development=# \d hash_test
                                    Table "public.hash_test"
    Column    |  Type   | Collation | Nullable |                       Default
--------------+---------+-----------+----------+-----------------------------------------------
 hash_test_id | integer |           | not null | nextval('hash_test_hash_test_id_seq'::regclass)
 hash_id      | integer |           |          |
 code         | text    |           |          |
 hash         | integer |           |          |
Indexes:
    "hash_test_pkey" PRIMARY KEY, btree (hash_test_id)
    "hash_test_hash_id_idx" btree (hash_id)
    "hash_test_hash_idx" btree (hash)
Foreign-key constraints:
    "hash_test_hash_id_fkey" FOREIGN KEY (hash_id) REFERENCES hashes(hash_id)
```

#rows in hashes = 5

#rows in hash_test =~ 13,300,000

#distinct hashes in hash_test =~ 13,300,000 / 5

Foreword on Complexities



Indices

Are good. We want. Mostly.

- Overhead presents itself in INSERT/UPDATE/DELETE costs of maintaining a balanced tree
- Makes searches O(log(M)) where M is the size of the table.

Node:

- An index scan node looks something like this:

```
Index Scan using hash_test_hash_idx on hash_test  (cost=0.56..11.84 rows=114 width=28) (actual time=0.009..0.024 rows=94 loops=1)
  Index Cond: ((hash >= 1) AND (hash <= 10))
```

- Usually when one encounters such a node, safe to move on.

The “opposite” of this is:

- A sequential scan of your table looks something like this:

```
Gather  (cost=1000.00..279692.06 rows=1 width=28) (actual time=665.159..665.179 rows=94 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Seq Scan on hash_test  (cost=0.00..278691.96 rows=1 width=28) (actual time=480.913..653.764 rows=31 loops=3)
        Filter: ((hash >= 1) AND (hash <= 10))
        Rows Removed by Filter: 4433302
```

- If by your estimates, your query should run faster, might be worth looking for a node that looks like that

Joins

Assuming 2 tables of sizes N and M rows, N < M.

Nested Loop:

- Worst case O(N*M)
- Usually a significantly small table + portion of an indexed larger table

```
sandbox_development=# explain analyze select * from hash_test join hashes using(hash_id) where algorithm = 'bumpy';
```

Hash:

- Worst case $O(N*h_c + M*h_m)$
- Since h_c and h_m are typically independent of input, O(N+M)
- Either missing index, or joins very large portion of bigger table

```
sandbox_development=# explain analyze select * from hash_test join hashes using(hash_id);
```

Merge:

- Worst case O(N+M)
- Joined on equality only
- Both “node inputs” sorted on join key

```
sandbox_development=# explain analyze select * from hash_test join hashes using(hash_id) order by hash_id limit 1000000;
```

Sequential Scans

Usually bad. This is usually a prime candidate for optimization.

Usually.

3 caveats:

- You need all the data from the table anyway

```
sandbox_development=# explain analyze select * from hash_test;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on hash_test  (cost=0.00..328553.73 rows=13296473 width=28) (actual time=0.012..1589.238 rows=13300001 loops=1)
 Planning time: 0.085 ms
 Execution time: 2333.155 ms
(3 rows)
```

- You need so much data from the table that indices are just an overhead (~ >8%)
- Your table is so small that indices are an overhead

```
sandbox_development=# explain analyze select * from hash_test join hashes using(hash_id) where algorithm in ('bumpy', 'collidey');
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=1.09..472821.55 rows=5318589 width=34) (actual time=0.074..3621.167 rows=5323082 loops=1)
   Hash Cond: (hash_test.hash_id = hashes.hash_id)
   ->  Seq Scan on hash_test  (cost=0.00..328553.73 rows=13296473 width=28) (actual time=0.012..1646.144 rows=13300001 loops=1)
   ->  Hash  (cost=1.06..1.06 rows=2 width=10) (actual time=0.014..0.014 rows=2 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on hashes  (cost=0.00..1.06 rows=2 width=10) (actual time=0.009..0.011 rows=2 loops=1)
               Filter: (algorithm = ANY ('{bumpy,collidey}'::text[]))
               Rows Removed by Filter: 3
 Planning time: 0.306 ms
 Execution time: 3925.430 ms
(10 rows)
```

Bitmap Index and Heap Scans

1 part slower than a pure index scan.
1 part faster than a sequential scan.
98 parts full awesome.

Bitmap Index + Heap scan (OR):

```
sandbox_development=# SET enable_seqscan = OFF;
SET
sandbox_development=# explain analyze select * from hash_test where hash_id = 1 or hash_id = 3;
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on hash_test  (cost=176598.31..452837.08 rows=4833118 width=28) (actual time=323.013..1498.157 rows=5319772 loops=1)
   Recheck Cond: ((hash_id = 1) OR (hash_id = 3))
   Rows Removed by Index Recheck: 2689820
   Heap Blocks: exact=64588 lossy=33033
   ->  BitmapOr  (cost=176598.31..176598.31 rows=5376651 width=0) (actual time=312.913..312.913 rows=0 loops=1)
         ->  Bitmap Index Scan on hash_test_hash_id_idx  (cost=0.00..86939.98 rows=2683672 width=0) (actual time=168.812..168.812 rows=2658453 loops=1)
               Index Cond: (hash_id = 1)
         ->  Bitmap Index Scan on hash_test_hash_id_idx  (cost=0.00..87241.78 rows=2692979 width=0) (actual time=144.099..144.099 rows=2661319 loops=1)
               Index Cond: (hash_id = 3)
 Planning time: 0.096 ms
 Execution time: 1800.150 ms
(11 rows)
```

Bitmap Heap scan (fetch optimization):

```
sandbox_development=# explain analyze with c as (select array[1,2,3] as a) select * from hash_test join c on hash_id = any(c.a);
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=296287.40..877611.61 rows=650063 width=60) (actual time=470.691..3190.652 rows=7982393 loops=1)
   CTE c
     ->  Result  (cost=0.00..0.01 rows=1 width=32) (actual time=0.001..0.001 rows=1 loops=1)
   ->  CTE Scan on c  (cost=0.00..0.02 rows=1 width=32) (actual time=0.003..0.004 rows=1 loops=1)
   ->  Bitmap Heap Scan on hash_test  (cost=296287.39..758923.83 rows=11868775 width=28) (actual time=470.680..2050.699 rows=7982393 loops=1)
         Recheck Cond: (hash_id = ANY (c.a))
         Rows Removed by Index Recheck: 1793542
         Heap Blocks: exact=64759 lossy=33033
         ->  Bitmap Index Scan on hash_test_hash_id_idx  (cost=0.00..293320.20 rows=11868775 width=0) (actual time=459.391..459.391 rows=7982393 loops=1)
               Index Cond: (hash_id = ANY (c.a))
 Planning time: 0.169 ms
 Execution time: 3651.267 ms
(12 rows)
```


Thank You
TOP TALENT AND TEAMWORK
ACCOUNTABLE FOR RESULTS
OPERATE AS AN OWNER
BEST ANSWER WINS