Why physical storage of your database tables might matter

Apoorva Aggarwal

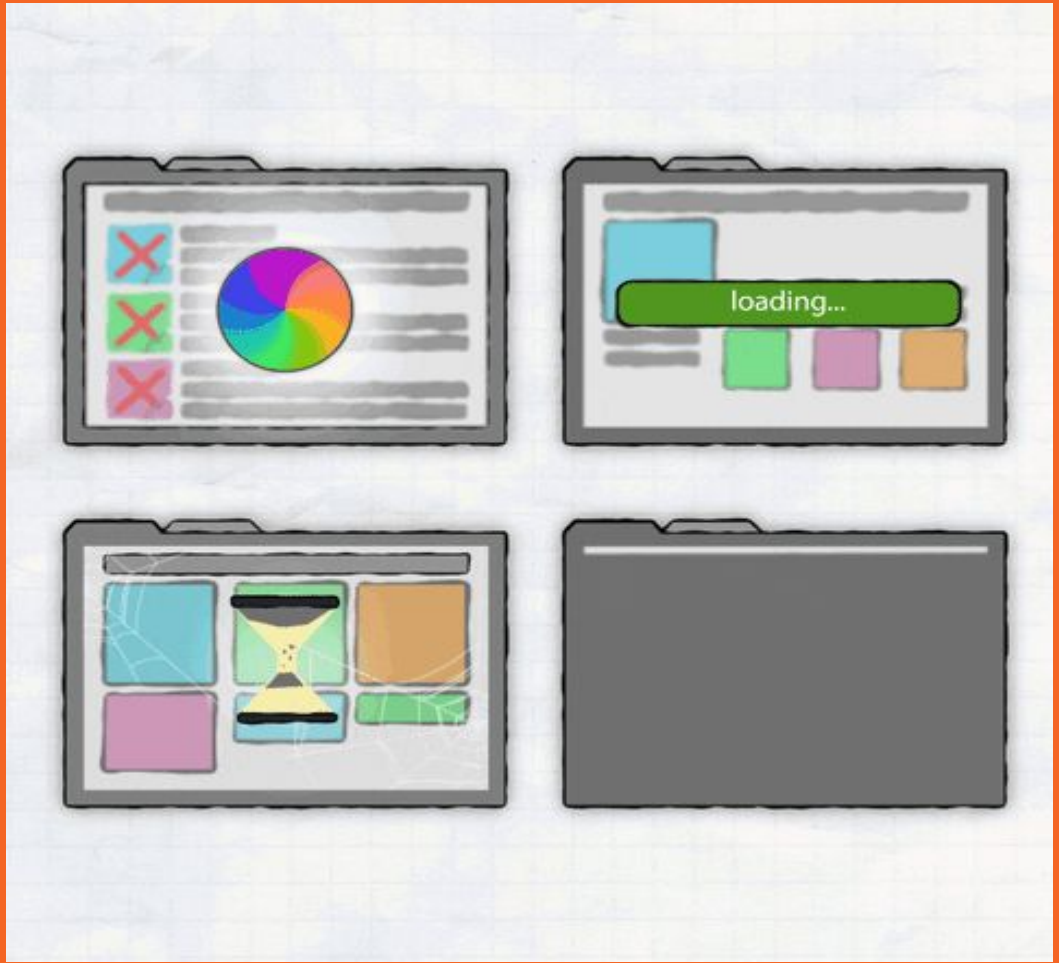
About Me

Data Platform Engineer at Grofers helping build systems for data informed decisions

@apoorva_1993

apoorvaaggarwal


It was a dark
and stormy night.
3-13


loading...

The War Zone

Bird's eye view

Behind the scenes

Narrowing down into problem space

```
db=> SELECT *
FROM personalized_recommendations
WHERE customer_id = 1;
...

db=> \d personalized_recommendations
           Table "public.personalized_recommendations"
    Column     |       Type       | Collation | Nullable | Default
---------------+------------------+-----------+----------+---------
 customer_id   | integer          |           | not null |
 product_id    | integer          |           | not null |
 score         | double precision |           | not null |
 ...
Indexes:
    "personalized_recommendations_temp_customer_id_idx1" btree
(customer_id)
```

We looked at the query plan

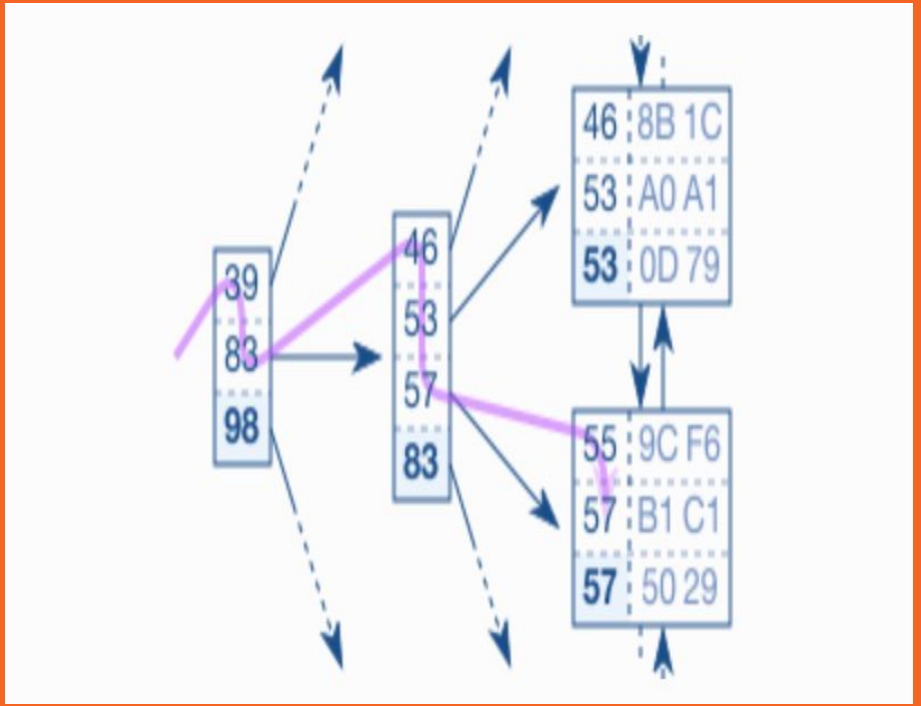
```
EXPLAIN ANALYZE
SELECT *
FROM personalized_recommendations
WHERE customer_id = 25001;


QUERY PLAN
- - - - - - - - - - - - - - -
Index Scan using personalized_recommendations_temp_customer_id_idx
on personalized_recommendations (cost=0.57..863.90 rows=214
width=38) (actual time=10.372..110.246 rows=201 loops=1)
 Index Cond: (customer_id = 25001)
Planning time: 0.066 ms
Execution time: 110.335 ms
```

Default index in postgres is of type BTree which constitutes a BTree or a balanced tree of index entries and index leaf nodes which store physical address of an index entry.

An index lookup requires three steps:

1. Tree traversal - O(log n)
2. Following the leaf node chain - O(n) ~ O(200)
3. Fetching the table data



Distribution of Data

In postgres, location of a row is given by ctid which is a tuple. ctid is of type tid (tuple identifier), called ItemPointer in the C code. Per documentation:

> *This is the data type of the system column ctid. A tuple ID is a pair (**block number**, **tuple index within block**) that identifies the physical location of the row within its table.*

```
customer_id | product_id | ctid
 - - - - - - -+ - - - - - - -
 1254 | 284670 | (3789,28)
 1254 | 18934 | (7071,73)
 1254 | 14795 | (8033,19)
 1254 | 10908 | (9591,60)
 1254 | 95032 | (11017,83)
 1254 | 318562 | (11134,65)
 1254 | 18854 | (11275,54)
 1254 | 109943 | (11827,76)
 1254 | 105 | (16309,104)
 1254 | 3896 | (18432,8)
 1254 | 3890 | (20062,90)
 1254 | 318550 | (20488,84)
 1254 | 37261 | (20585,62)
 ...
```

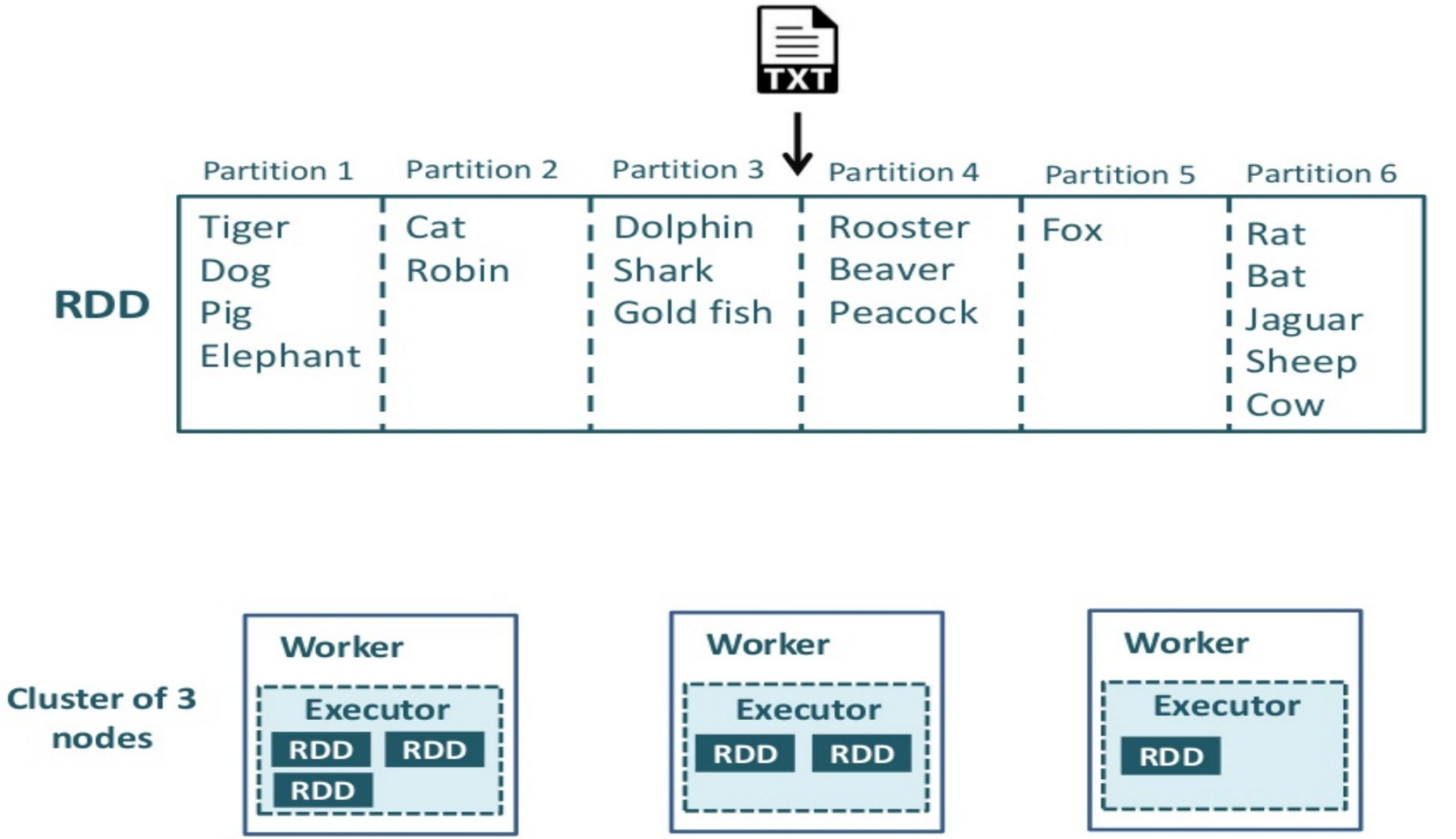
Possible solutions to root cause and exploring feasibility

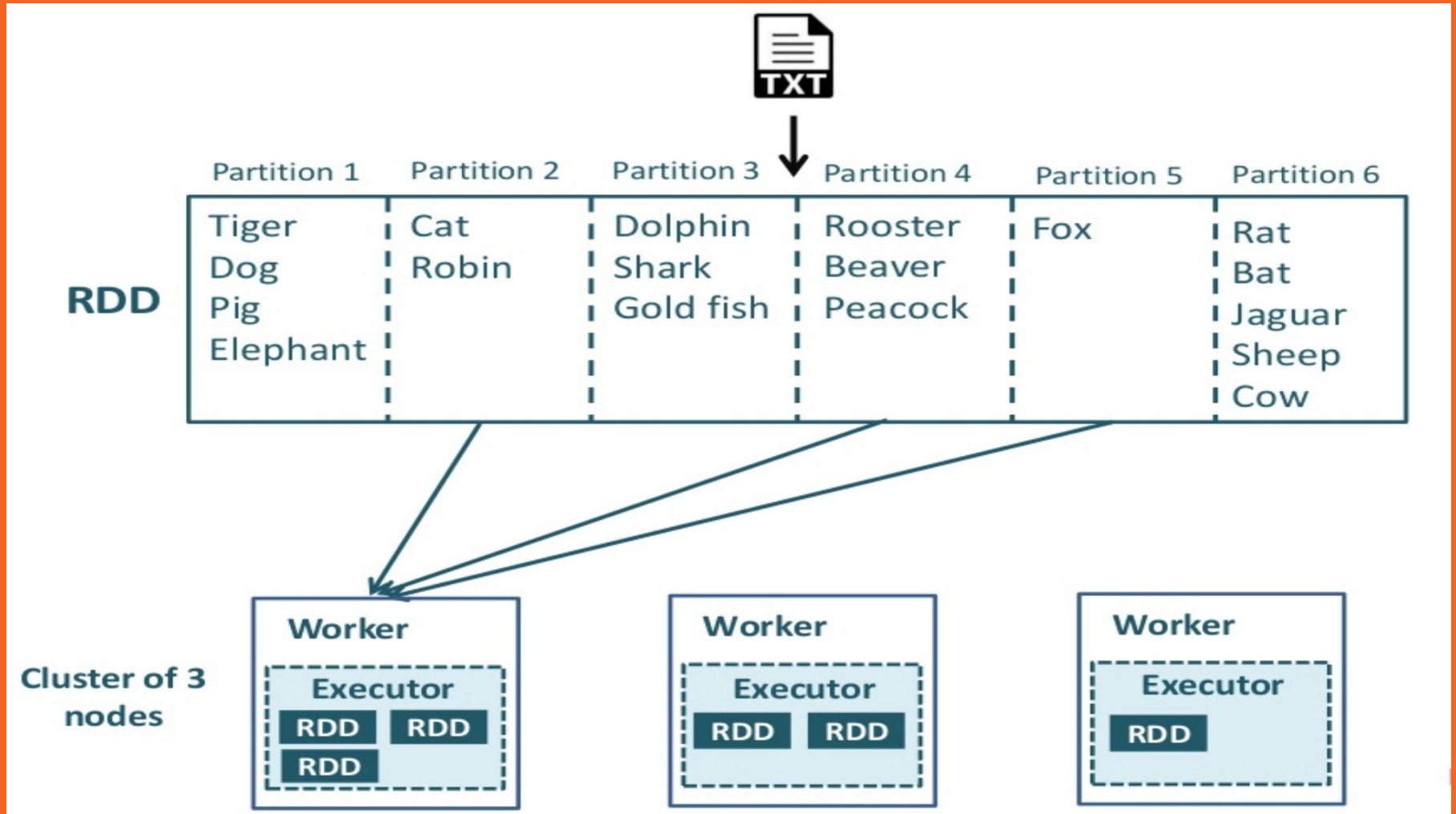
CLUSTER command

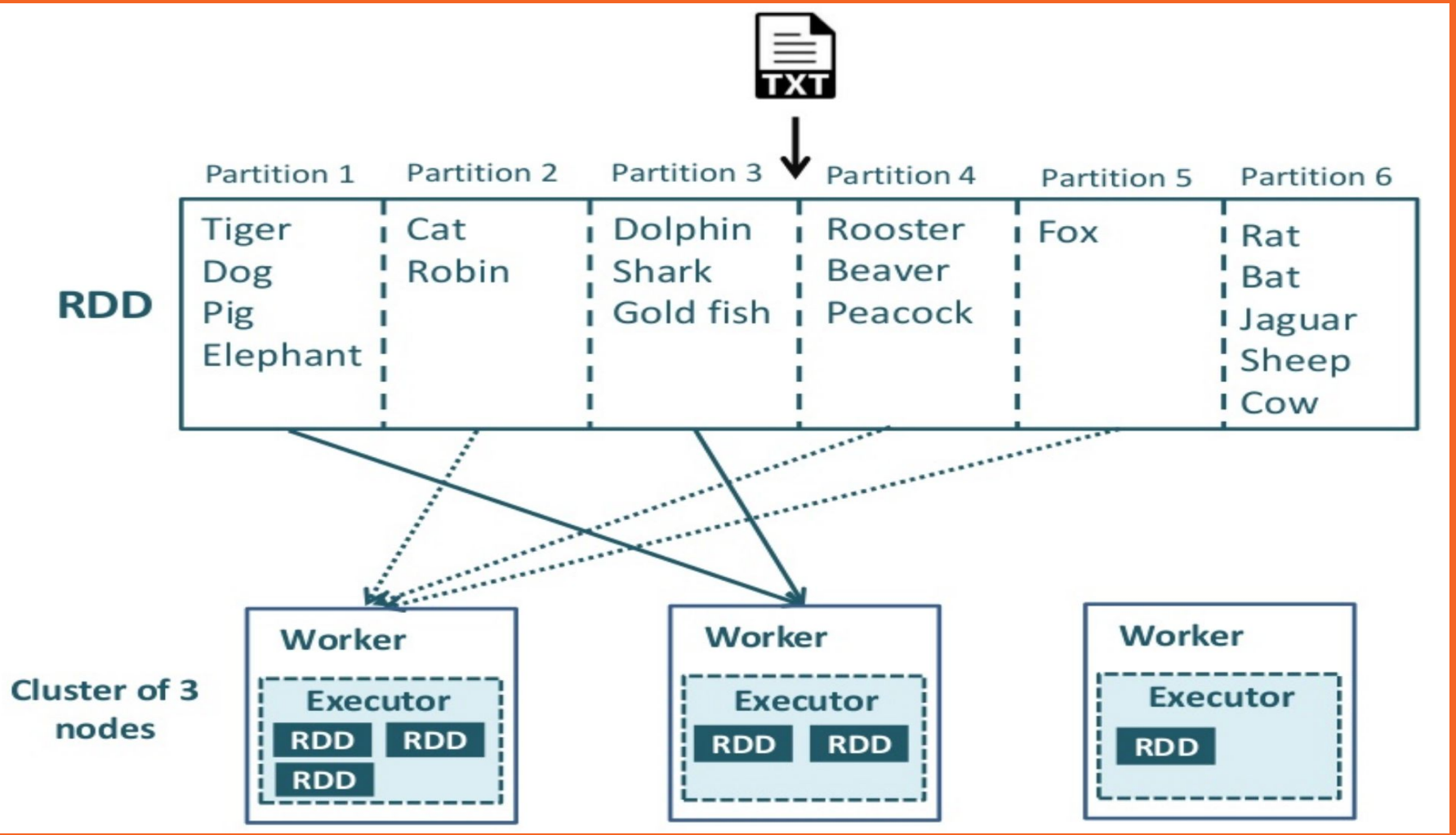
- Physically rearranges the rows on disk based on a given column.
- Acquiring a READ WRITE lock on table and requiring 2.5x the table size made it tricky to use.

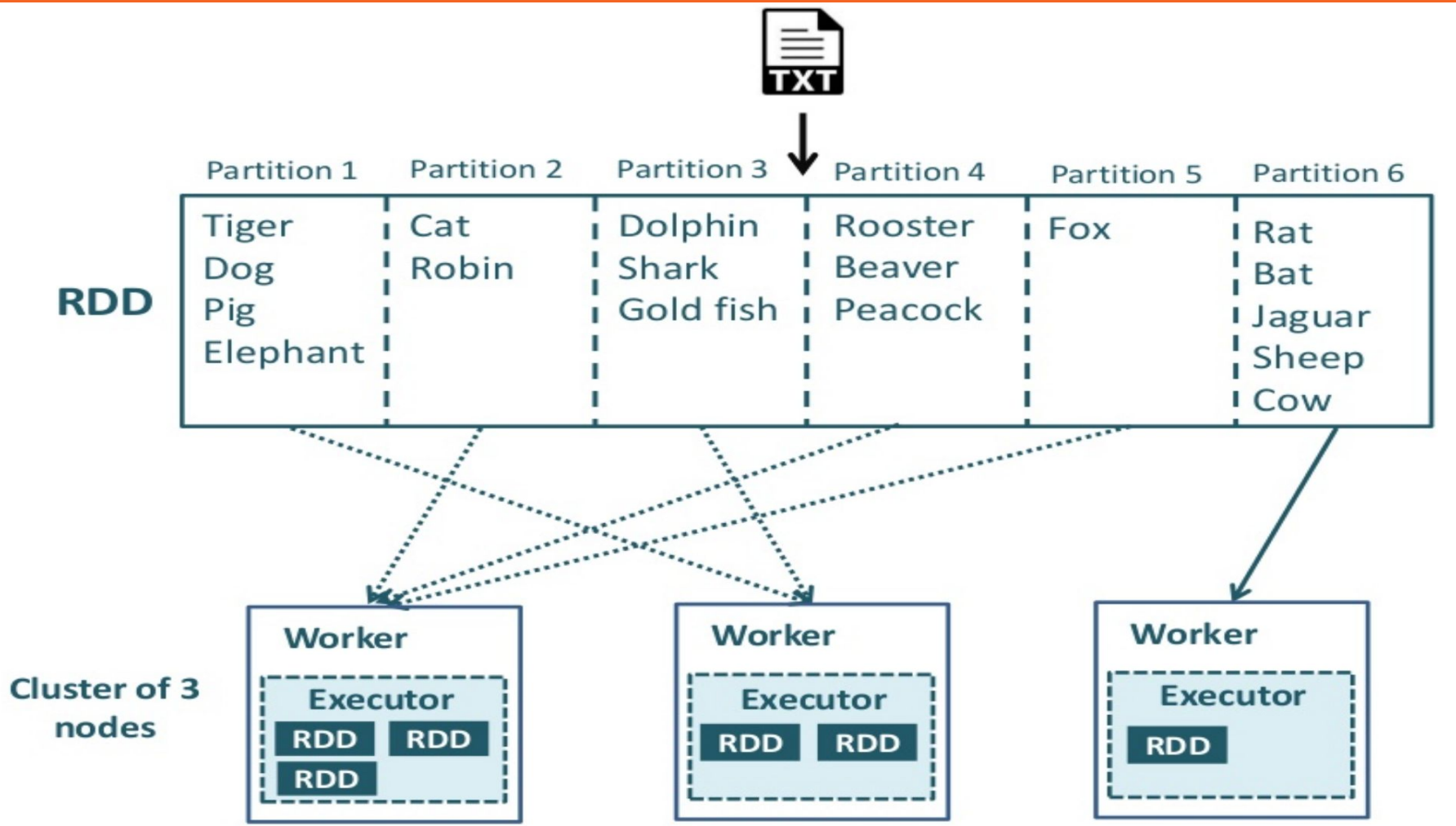
An attempt to fix the problem right from the source

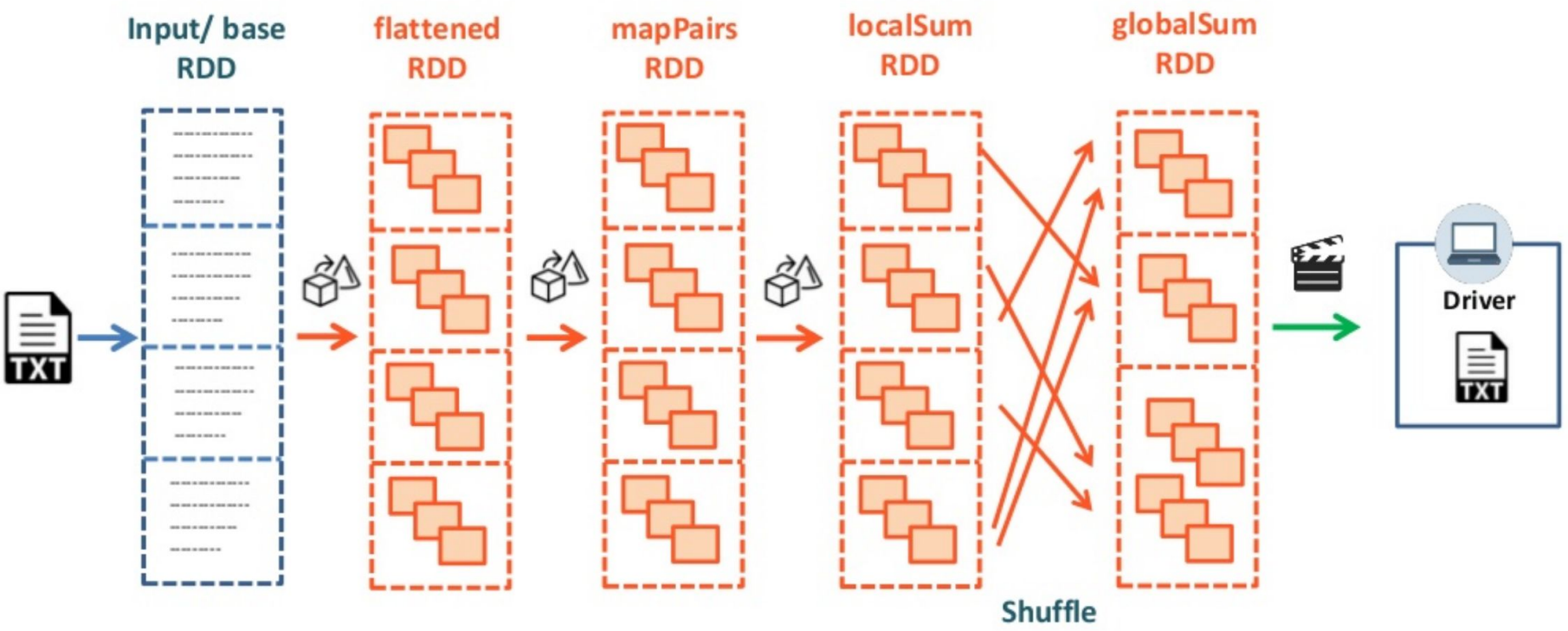
Understanding how Spark writes data to a table


TXT
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
Partition 6
RDD
Tiger
Dog
Pig
Elephant
Cat
Robin
Dolphin
Shark
Gold fish
Rooster
Beaver
Peacock
Fox
Rat
Bat
Jaguar
Sheep
Cow
Cluster of 3 nodes
Worker
Executor
RDD
RDD
RDD
Worker
Executor
RDD
RDD
Worker
Executor
RDD


TXT
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
Partition 6
RDD
Tiger
Dog
Pig
Elephant
Cat
Robin
Dolphin
Shark
Gold fish
Rooster
Beaver
Peacock
Fox
Rat
Bat
Jaguar
Sheep
Cow
Cluster of 3 nodes
Worker
Executor
RDD
RDD
RDD
Worker
Executor
RDD
RDD
Worker
Executor
RDD


TXT
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
Partition 6
RDD
Tiger
Dog
Pig
Elephant
Cat
Robin
Dolphin
Shark
Gold fish
Rooster
Beaver
Peacock
Fox
Rat
Bat
Jaguar
Sheep
Cow
Cluster of 3 nodes
Worker
Executor
RDD
RDD
RDD
Worker
Executor
RDD
RDD
Worker
Executor
RDD


TXT
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
Partition 6
RDD
Tiger
Dog
Pig
Elephant
Cat
Robin
Dolphin
Shark
Gold fish
Rooster
Beaver
Peacock
Fox
Rat
Bat
Jaguar
Sheep
Cow
Cluster of 3 nodes
Worker
Executor
RDD
RDD
RDD
Worker
Executor
RDD
RDD
Worker
Executor
RDD


Input/ base RDD
flattened RDD
mapPairs RDD
localSum RDD
globalSum RDD
TXT
Shuffle
Driver
TXT

Looking at the code we found that we had partitioned on product_id for a particular transformation operation.

Note how rows containing a particular product ID are in one partition

Testing hypothesis

```
SELECT *, ctid FROM personalized_recommendations WHERE product_id =
284670
 product_id | customer_id |   ctid
------------+-------------+----------
     284670 |        1133 | (479502,71)
     284670 |        2488 | (479502,72)
     284670 |        3657 | (479502,73)
     284670 |        2923 | (479502,74)
     284670 |        6911 | (479502,75)
     284670 |        9018 | (479502,76)
     284670 |        4263 | (479502,77)
     284670 |        1331 | (479502,78)
     284670 |        3242 | (479502,79)
     284670 |        3661 | (479502,80)
     284670 |        9867 | (479502,81)
     284670 |        7066 | (479502,82)
     284670 |       10267 | (479502,83)
     284670 |        7499 | (479502,84)
     284670 |        8011 | (479502,85)
```

Attempting to align the data

We repartitioned the dataframe

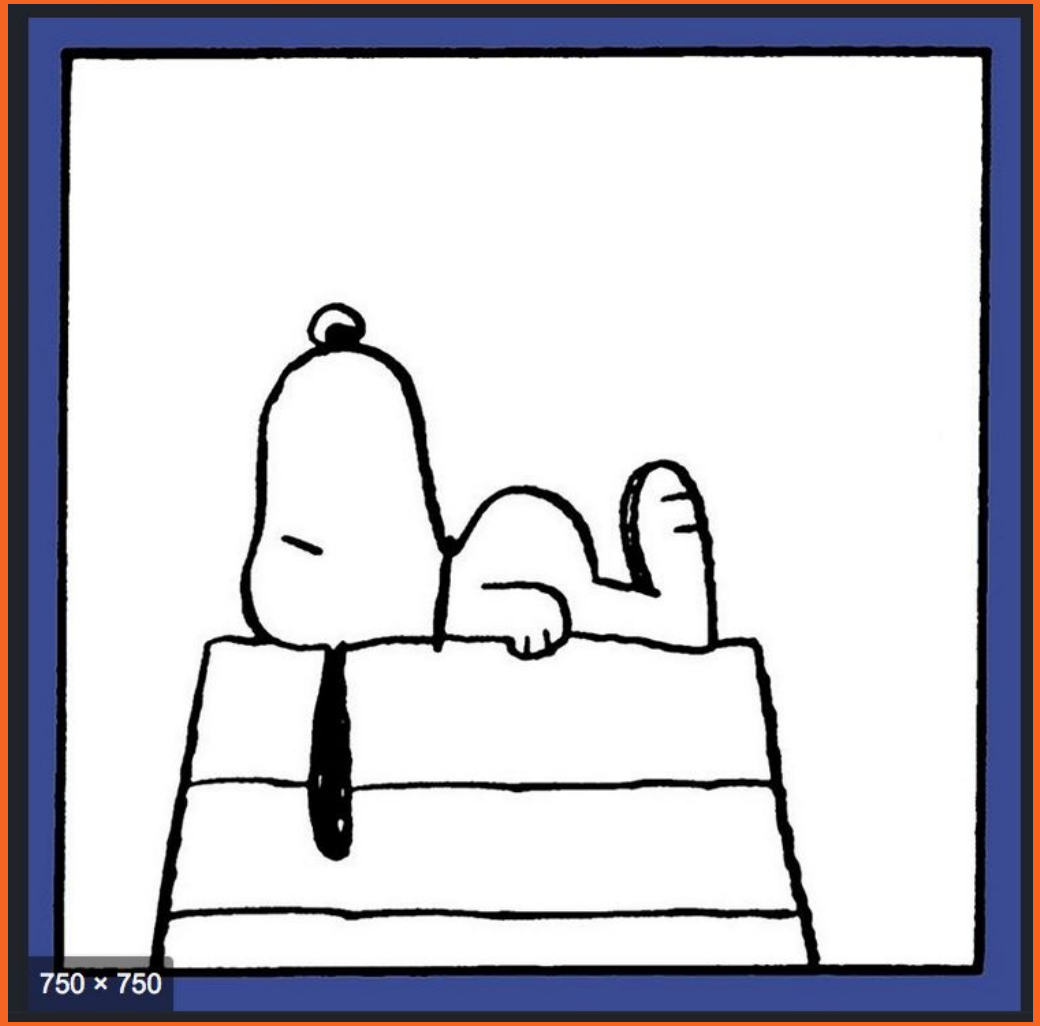
```
df.repartition($”customer_id”)
```

New data distribution

Alas, the table is still not pivoted on customer_id.

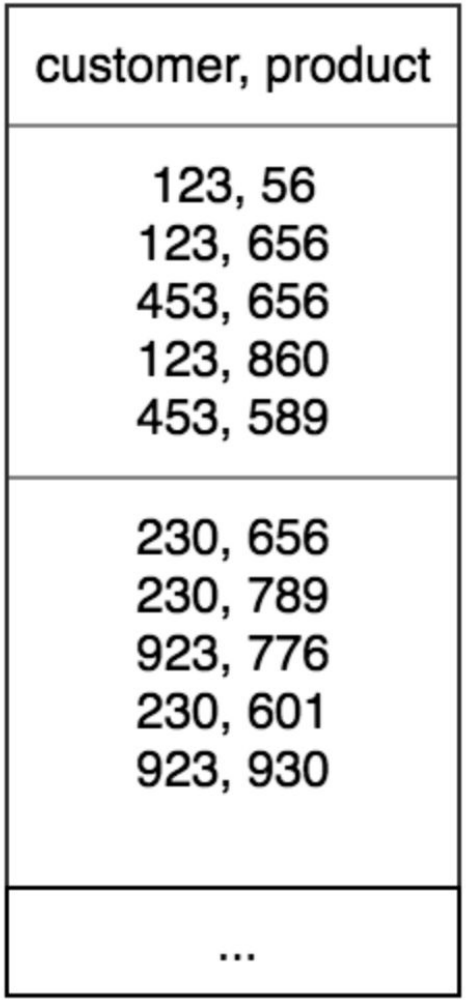
What did we do wrong?

```
db=> SELECT product_id,customer_id,ctid FROM
personalized_recommendations WHERE customer_id = 28460
limit 20;
 customer_id | product_id | ctid
-------------+------------+---------
 28460 | 1133 | (0,24)
 28460 | 2488 | (4,7)
 28460 | 3657 | (9,83)
 28460 | 2923 | (18,54)
 28460 | 6911 | (20,42)
 28460 | 9018 | (31,59)
 28460 | 4263 | (35,79)
 28460 | 1331 | (38,14)
 28460 | 3242 | (40,41)
 28460 | 3661 | (55,105)
 28460 | 9867 | (57,21)
 28460 | 7066 | (61,28)
 28460 | 10267 | (62,63)
 28460 | 7499 | (66,8)
```


750 × 750

Understanding Repartitioning

Visualizing the dataframe

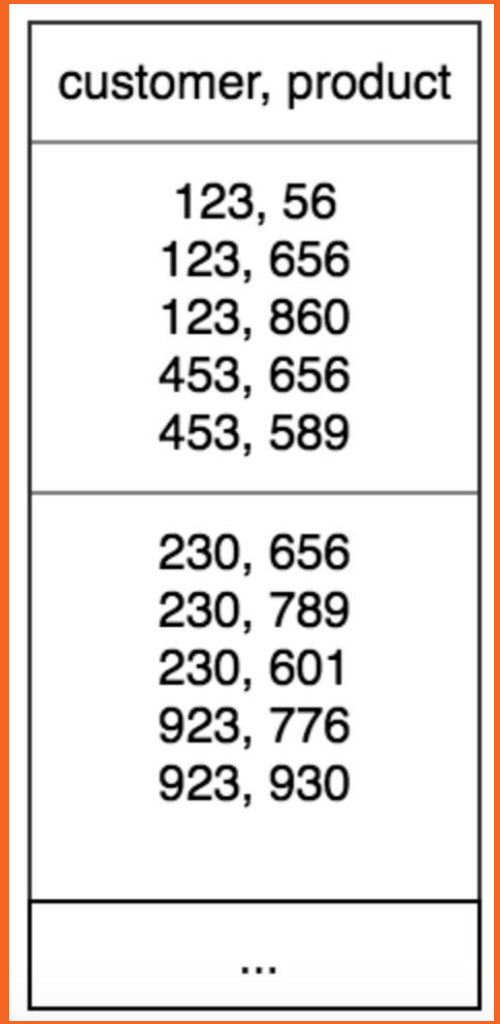


Bring all rows of a product together?
How?

Sort them maybe!?

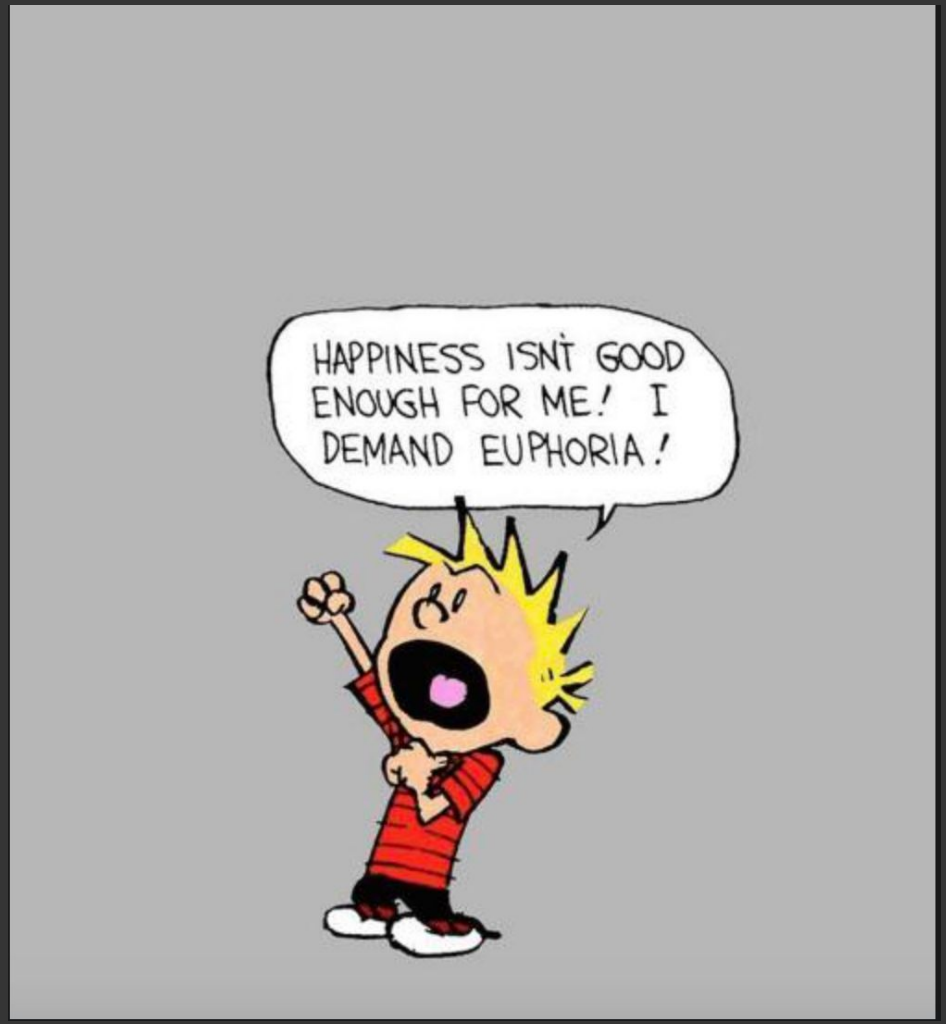
```
df.repartition($"customer_id").sortWithinPartitions($"customer_id")
```

After sorting rows within a partition by customer_id



Let’s write this to the database table
And see distribution

```
customer_id | product_id | ctid
 - - - - - - -+ - - - - - - - + - - - - - -
 1254 | 284670 | (212,95)
 1254 | 18854 | (212,96)
 1254 | 18850 | (212,97)
 1254 | 318560 | (212,98)
 1254 | 318562 | (212,99)
 1254 | 318561 | (212,100)
 1254 | 10732 | (212,101)
 1254 | 108 | (212,102)
 1254 | 11237 | (212,103)
 1254 | 318058 | (212,104)
 1254 | 38282 | (212,105)
 1254 | 3884 | (212,106)
 1254 | 31 | (212,107)
 1254 | 318609 | (215,1)
 1254 | 2 | (215,2)
 1254 | 240846 | (215,3)
 1254 | 197964 | (215,4)
 1254 | 232970 | (215,5)
 1254 | 124472 | (215,6)
 1254 | 19481 | (215,7)
 …
```


HAPPINESS ISNT GOOD ENOUGH FOR ME! I DEMAND EUPHORIA!

Testing the solution

The final test still remains. Will the query execution now be faster. Let’s see what the query planner says

```
EXPLAIN ANALYZE
SELECT *
FROM personalized_recommendations
WHERE customer_id = 25001;


QUERY PLAN


Bitmap Heap Scan on
personalized_recommendations(cost=66.87..13129.94 rows=3394
width=38) (actual time=2.843..3.259 rows=201 loops=1)
 Recheck Cond: (customer_id = 25001)
 Heap Blocks: exact=2
 -> Bitmap Index Scan on
personalized_recommendations_temp_customer_id_idx (cost=0.00..66.02
rows=3394 width=0) (actual time=1.995..1.995 rows=201 loops=1)
 Index Cond: (customer_id = 25001)
 Planning time: 0.067 ms
 Execution time: 3.322 ms
```

Why a bitmap scan

A plain index scan fetches one tuple-pointer at a time from the index, and immediately visits that tuple in the table.
A bitmap scan fetches all the tuple-pointers from the index in one go, sorts them using an in-memory "bitmap" data structure, and then visits the table tuples in physical tuple-location order.

```
customer_id | product_id | ctid
 - - - - - - -+ - - - - - - - + - - - - - -
 1254 | 284670 | (212,95)
 1254 | 18854 | (212,96)
 1254 | 18850 | (212,97)
 1254 | 318560 | (212,98)
 1254 | 318562 | (212,99)
 1254 | 318561 | (212,100)
 1254 | 10732 | (212,101)
 1254 | 108 | (212,102)
 1254 | 11237 | (212,103)
 1254 | 318058 | (212,104)
 1254 | 38282 | (212,105)
 1254 | 3884 | (212,106)
 1254 | 31 | (212,107)
 1254 | 318609 | (215,1)
 1254 | 2 | (215,2)
 1254 | 240846 | (215,3)
 1254 | 197964 | (215,4)
 1254 | 232970 | (215,5)
 1254 | 124472 | (215,6)
 1254 | 19481 | (215,7)
 …
```

—

Execution time came down from ~100 ms to ~3ms.

We are hiring!

Email: apoorva.aggarwal@grofers.com