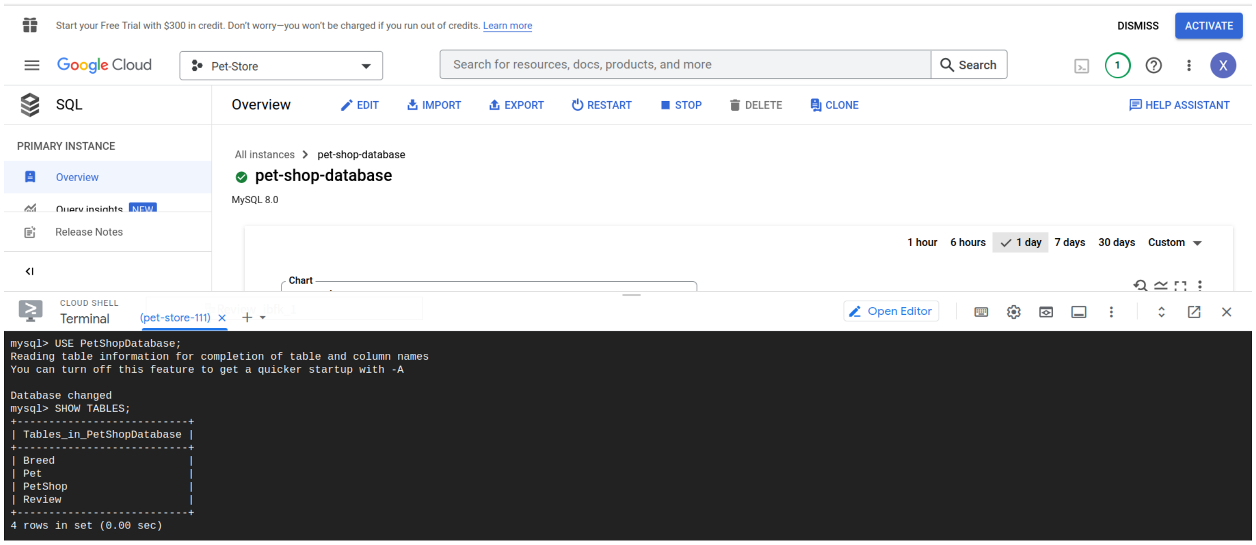
Part 1 Database Implementation:



```
mysql> USE PetShopDatabase;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+---------------------------+
| Tables_in_PetShopDatabase |
+---------------------------+
| Breed                     |
| Pet                       |
| PetShop                   |
| Review                    |
+---------------------------+
4 rows in set (0.00 sec)
```

We created the database locally and imported that into GCP. All of us are able to connect to our database using MySQL Workbench.

Our DDL commands

```
CREATE DATABASE PetShopDatabase DEFAULT CHARACTER SET 'utf8';

-- Add tables to PetShopDatabase only.
USE PetShopDatabase;

CREATE TABLE PetShop (
    `name` VARCHAR(50),
    location VARCHAR(100),

    PRIMARY KEY (`name`)
);

CREATE TABLE Breed (
    `name` VARCHAR(50),
    typical_height REAL CHECK (typical_height > 0),
    typical_weight REAL CHECK (typical_weight > 0),
    typical_age REAL CHECK (typical_age > 0),
    description VARCHAR(1000),
    image_url VARCHAR(200),
    type_name VARCHAR(50),

    PRIMARY KEY (`name`)
);

CREATE TABLE Pet (
```

```sql
    pet_id INT,
    `name` VARCHAR(50),
    sex VARCHAR(10) CHECK (sex = 'male' OR sex = 'female'),
    height REAL CHECK (height > 0),
    weight REAL CHECK (weight > 0),
    date_of_birth DATE,
    color VARCHAR(20),
    favorite_food VARCHAR(50),
    description VARCHAR(1000),
    price REAL CHECK (price >= 0),
    available BOOL,
    image_url VARCHAR(200),
    pet_shop_name VARCHAR(50),
    breed_name VARCHAR(50),

    PRIMARY KEY (pet_id),

    -- ON DELETE CASCADE ensures many-to-one relation.
    FOREIGN KEY (pet_shop_name) REFERENCES PetShop(`name`) ON DELETE CASCADE,
    FOREIGN KEY (breed_name) REFERENCES Breed(`name`) ON DELETE CASCADE
);

CREATE TABLE Review (
    review_id INT,
    rating INT CHECK (rating >= 1 AND rating <= 5),  -- 1 to 5 rating.
    review_date DATE DEFAULT (CURRENT_DATE),
    content VARCHAR(1000),
    customer_email VARCHAR(100),
    pet_shop_name VARCHAR(50),

    PRIMARY KEY (review_id),
    -- TODO: uncomment the following line after we add the Customer table.
    -- FOREIGN KEY (customer_email) REFERENCES Customer(email),
    FOREIGN KEY (pet_shop_name) REFERENCES PetShop(`name`) ON DELETE CASCADE
);

-- Reference:
https://stackoverflow.com/questions/20744508/best-way-to-manage-row-expiration-in-mysql
-- Clears the expired reservation. Starts just after midnight every day.
CREATE EVENT clearExpiredReservation
ON SCHEDULE EVERY 1 DAY
STARTS (CURRENT_DATE() + INTERVAL 1 DAY + INTERVAL 1 MINUTE)
DO
    DELETE FROM Reservation
    WHERE expire_date < CURRENT_DATE();

-- Make sure that the birth date of an incoming pet is valid. If it is not, set
-- to NULL.
CREATE TRIGGER checkPetDateOfBirth
```

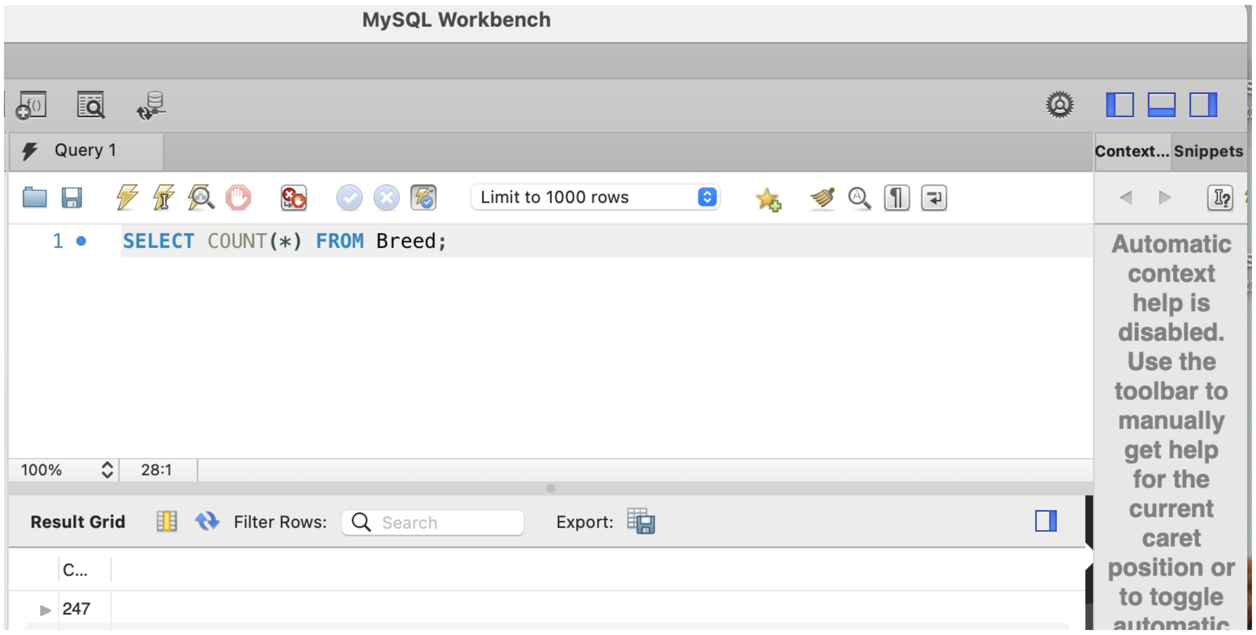
```
BEFORE INSERT ON Pet
FOR EACH ROW BEGIN
    IF NEW.date_of_birth > CURRENT_DATE() THEN
        SET NEW.date_of_birth = NULL;
    END IF;
END;

-- Make sure that the review_date of an incoming review is valid. If it is not,
-- set to NULL.
CREATE TRIGGER checkReviewDate
BEFORE INSERT ON Review
FOR EACH ROW BEGIN
    IF NEW.review_date > CURRENT_DATE() THEN
        SET NEW.review_date = NULL;
    END IF;
END;
```

Data import

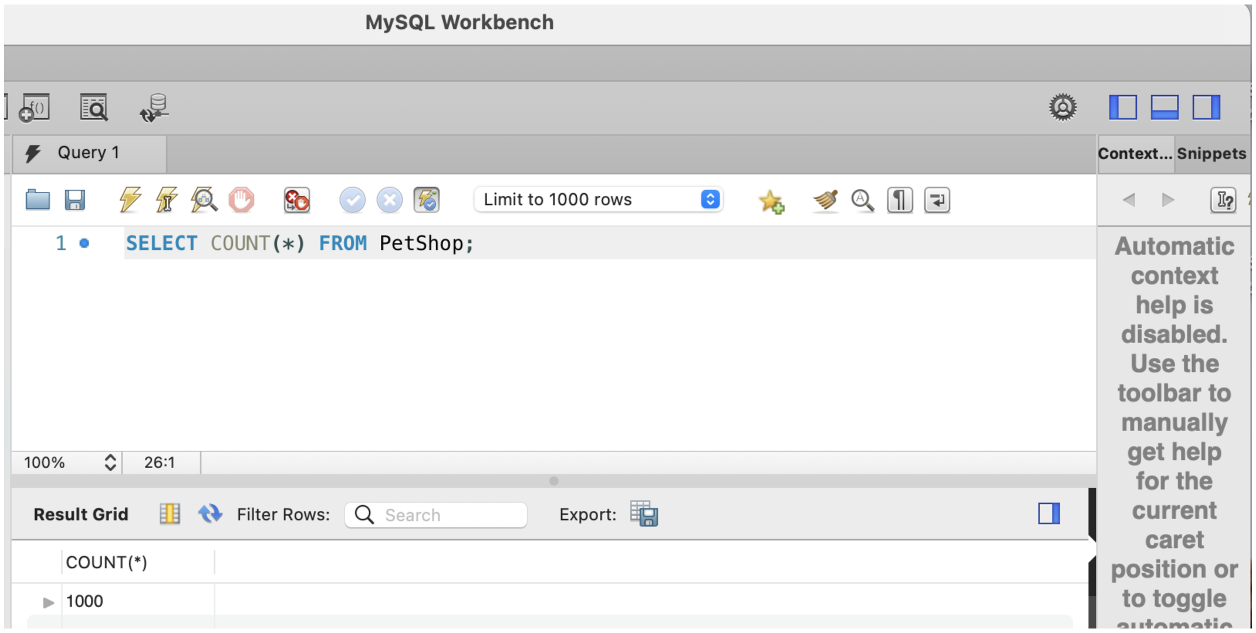
For the breed table, we use real data from https://data.world/len/dog-canine-breed-size-akc/workspace/file?filename=AKC+Breed+Info.csv to fill in the typical height and weights for different breeds of dogs. However, we are only able to make it to 247 rows, since there are not enough breeds of pets.

For the rest of the table: PetShop, Pet, and Review tables, we use randomly generated data. These three tables have 1000, 20000 and 50000 rows respectively, which means that on average there are 20 pets and 50 reviews for each pet shop.


MySQL Workbench
Query 1
Context...
Snippets
Limit to 1000 rows
100%
28:1
Result Grid
Filter Rows:
Search
Export:
Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic

```
SELECT COUNT(*) FROM Breed;
```

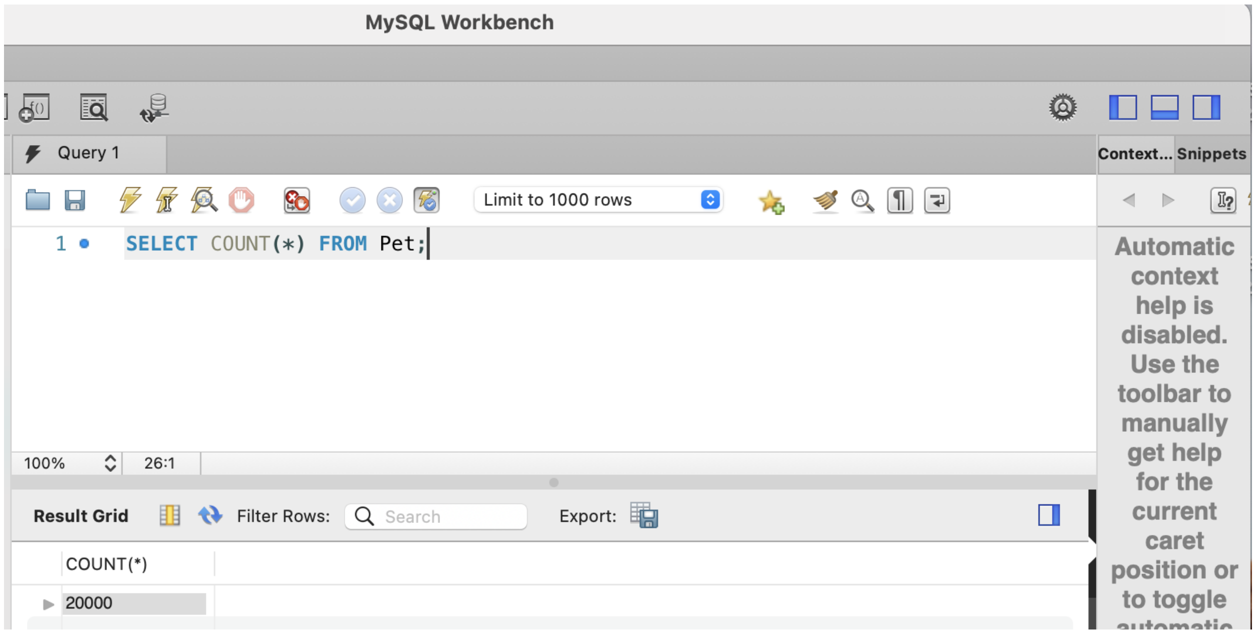
```
C...
247
```



```
SELECT COUNT(*) FROM PetShop;
```

```
COUNT(*)
1000
```

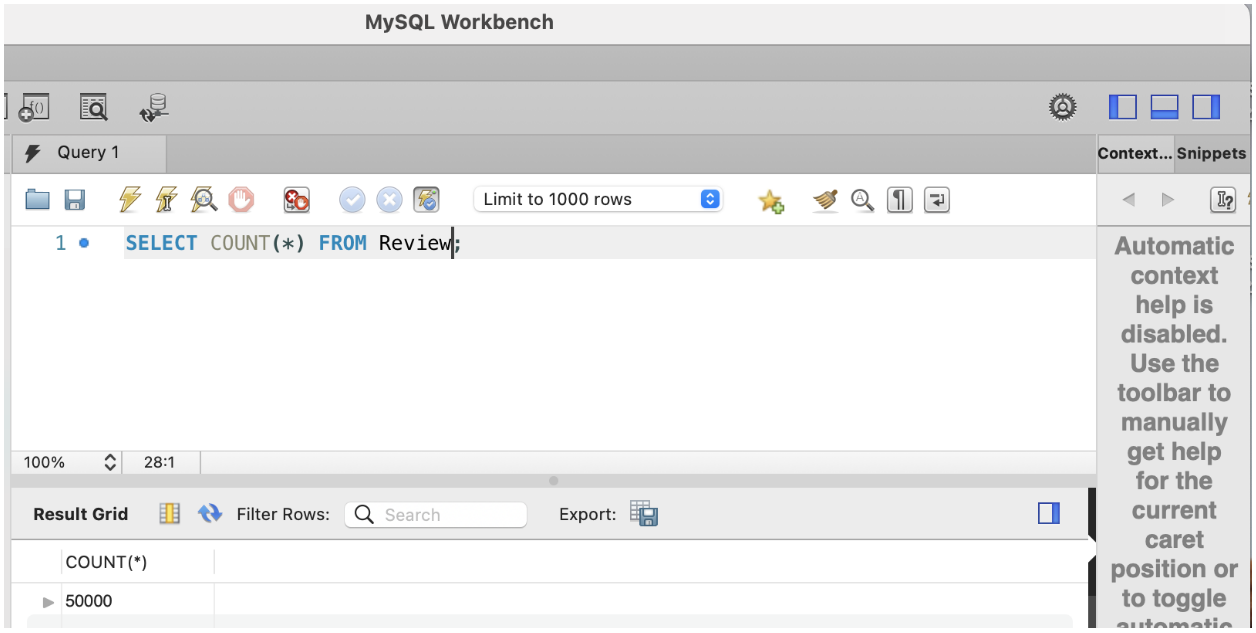
Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle



```
SELECT COUNT(*) FROM Pet;
```

```
COUNT(*)
20000
```

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle



```
SELECT COUNT(*) FROM Review;
```

```
COUNT(*)
50000
```

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle

Part 2 Advanced queries and indexing

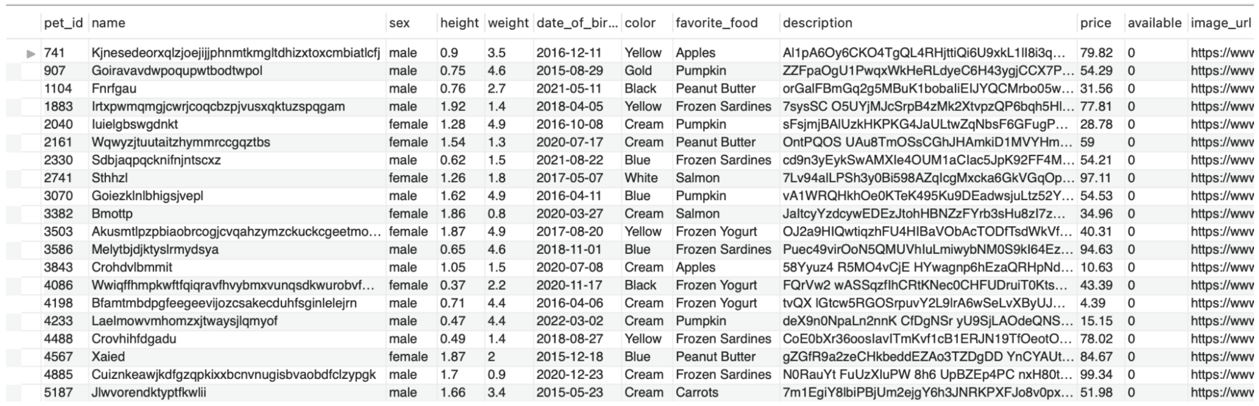
We tried out two advanced SQL queries:

1. **Find information of all big dogs (i.e. the dogs with a breed that has a typical height above average) that have an average lifespan of more than 10 years**

Query

```
SELECT *
FROM Pet JOIN Breed ON(Pet.breed_name = Breed.name)
WHERE breed_name IN (SELECT name
                       FROM Breed
                       WHERE typical_age > 10 AND
                       typical_height > (SELECT AVG(typical_height) FROM Breed))
;
```

Results



pet_id	name	sex	height	weight	date_of_bir...	color	favorite_food	description	price	available	image_url
741	Kjnesedeorxqlzjoejijjphnmtkmgltdhizxtoxcmbiatlcfj	male	0.9	3.5	2016-12-11	Yellow	Apples	Al1pA6Oy6CKO4TgQL4RHjttiQi6U9xkL1Il8i3q...	79.82	0	https://wwv
907	Goiravavdwpoqupwtbodtwpol	male	0.75	4.6	2015-08-29	Gold	Pumpkin	ZZFpaOgU1PwqxWkHeRLdyeC6H43ygjCCX7P...	54.29	0	https://wwv
1104	Fnrfgau	male	0.76	2.7	2021-05-11	Black	Peanut Butter	orGalFBmGq2g5MBuK1bobaliEIJYQCMrbo05w...	31.56	0	https://wwv
1883	Irtxpwmqmgjcwrjcoqcbzpjvusxqktuzspqgam	male	1.92	1.4	2018-04-05	Yellow	Frozen Sardines	7sysSC O5UYjMJcSrpB4zMk2XtvpzQP6bqh5Hl...	77.81	0	https://wwv
2040	Iuielgbswgdnkt	female	1.28	4.9	2016-10-08	Cream	Pumpkin	sFsjmjBAlUzkHKPKG4JaULtwZqNbsF6GFugP...	28.78	0	https://wwv
2161	Wqwyzjtuutaitzhymmrccgqztbs	female	1.54	1.3	2020-07-17	Cream	Peanut Butter	OntPQOS UAu8TmOSsCGhJHAmkiD1MVYHm...	59	0	https://wwv
2330	Sdbjaqpqcknifnjntscxz	male	0.62	1.5	2021-08-22	Blue	Frozen Sardines	cd9n3yEykSwAMXIe4OUM1aClac5JpK92FF4M...	54.21	0	https://wwv
2741	Sthhzl	female	1.26	1.8	2017-05-07	White	Salmon	7Lv94alLPSh3y0Bi598AZqIcgMxcka6GkVGqOp...	97.11	0	https://wwv
3070	Goiezklnlbhigsjvepl	male	1.62	4.9	2016-04-11	Blue	Pumpkin	vA1WRQHkhOe0KTeK495Ku9DEadwsjuLtz52Y...	54.53	0	https://wwv
3382	Bmottp	female	1.86	0.8	2020-03-27	Cream	Salmon	JaltcyYzdcywEDEzJtohHBNZzFYrb3sHu8zI7z...	34.96	0	https://wwv
3503	Akusmtlpzpbiaobrcogjcvqahzymzckuckcgeetmo...	female	1.87	4.9	2017-08-20	Yellow	Frozen Yogurt	OJ2a9HIQwtiqzhFU4HIBaVObAcTODfTsdWkVf...	40.31	0	https://wwv
3586	Melytbjdjktyslrmydsya	male	0.65	4.6	2018-11-01	Blue	Frozen Sardines	Puec49virOoN5QMUVhIuLmiwybNM0S9kI64Ez...	94.63	0	https://wwv
3843	Crohdvlbmmit	male	1.05	1.5	2020-07-08	Cream	Apples	58Yyuz4 R5MO4vCjE HYwagnp6hEzaQRHpNd...	10.63	0	https://wwv
4086	Wwiqffhmpkwftfqiqravfhvybmxvunqsdkwurobvf...	female	0.37	2.2	2020-11-17	Black	Frozen Yogurt	FQrVw2 wASSqzfIhCRtKNec0CHFUDruiT0Kts...	43.39	0	https://wwv
4198	Bfamtmbdpgfeegeevijozcsakecduhfsginlelejrn	male	0.71	4.4	2016-04-06	Cream	Frozen Yogurt	tvQX lGtcw5RGOSrpuvY2L9lrA6wSeLvXByUJ...	4.39	0	https://wwv
4233	Laelmowvmhomzxjtwaysjlqmyof	male	0.47	4.4	2022-03-02	Cream	Pumpkin	deX9n0NpaLn2nnK CfDgNSr yU9SjLAOdeQNS...	15.15	0	https://wwv
4488	Crovhihfdgadu	male	0.49	1.4	2018-08-27	Yellow	Frozen Sardines	CoE0bXr36ooslavITmKvf1cB1ERJN19TfOeotO...	78.02	0	https://wwv
4567	Xaied	female	1.87	2	2015-12-18	Blue	Peanut Butter	gZGfR9a2zeCHkbeddEZAo3TZDgDD YnCYAUt...	84.67	0	https://wwv
4885	Cuiznkeawjkdfgzqpkixxbcnvnugisbvaobdfclzypgk	male	1.7	0.9	2020-12-23	Cream	Frozen Sardines	N0RauYt FuUzXluPW 8h6 UpBZEp4PC nxH80t...	99.34	0	https://wwv
5187	Jlwvorendktyptfkwlii	male	1.66	3.4	2015-05-23	Cream	Carrots	7m1EgiY8lbiPBjUm2ejgY6h3JNRKPXFJo8v0px...	51.98	0	https://wwv

Index1: on pet.breed_name

The step that takes the most of the time is the nested loop inner join, which joins the Pet and Breed table. Therefore to speed up the query, I first tried to add indices on the join attributes.

This index is defined as: *CREATE INDEX BREEDNAMEINDEX ON Pet(breed_name)*;
After adding this index, EXPLAIN ANALYSIS gives these outputs:

```
-> Nested loop inner join  (cost=790.54 rows=2173) (actual time=0.487..17.273
rows=5177 loops=1)
    -> Nested loop inner join  (cost=30.07 rows=27) (actual time=0.212..0.632
rows=65 loops=1)
        -> Filter: ((Breed.typical_age > 10) and (Breed.typical_height > (select
#3)))  (cost=20.46 rows=27) (actual time=0.189..0.371 rows=65 loops=1)
            -> Table scan on Breed  (cost=20.46 rows=247) (actual
time=0.077..0.191 rows=247 loops=1)
            -> Select #3 (subquery in condition; run only once)
                -> Aggregate: avg(Breed.typical_height)  (cost=50.65 rows=247)
(actual time=0.097..0.097 rows=1 loops=1)
                    -> Table scan on Breed  (cost=25.95 rows=247) (actual
time=0.028..0.080 rows=247 loops=1)
        -> Single-row index lookup on Breed using PRIMARY (name=Breed.`name`)
(cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=65)
    -> Index lookup on Pet using BREEDNAMEINDEX (breed_name=Breed.`name`)
(cost=20.09 rows=79) (actual time=0.097..0.250 rows=80 loops=65)
```

I used this index because we don't need to scan the whole pet table. Instead, we can use an index and this indexing actually reduces our cost of joining the two tables from around 810 originally to 790 now.

Index2: on breed.typical_height

```
-> Nested loop inner join  (cost=952.01 rows=2613) (actual time=0.240..17.341
rows=5177 loops=1)
```

```
    -> Nested loop inner join  (cost=37.50 rows=33) (actual time=0.046..0.437
rows=65 loops=1)
        -> Filter: ((Breed.typical_age > 10) and (Breed.typical_height > (select
#3)))  (cost=25.95 rows=33) (actual time=0.034..0.202 rows=65 loops=1)
            -> Table scan on Breed  (cost=25.95 rows=247) (actual
time=0.031..0.127 rows=247 loops=1)
            -> Select #3 (subquery in condition; run only once)
                -> Aggregate: avg(Breed.typical_height)  (cost=50.65 rows=247)
(actual time=0.087..0.087 rows=1 loops=1)
                    -> Index scan on Breed using heightIndex  (cost=25.95
rows=247) (actual time=0.026..0.070 rows=247 loops=1)
        -> Single-row index lookup on Breed using PRIMARY (name=Breed.`name`)
(cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=65)
    -> Index lookup on Pet using BREEDNAMEINDEX (breed_name=Breed.`name`)
(cost=20.04 rows=79) (actual time=0.102..0.255 rows=80 loops=65)
```

Here, in the WHERE clause, the conditions such as > will allow the database to filter many of the rows from the table and go through a small number of rows to return the required results. Therefore, I used indexing by adding these columns to the index. Here, using heightIndex actually does not change the cost. I think this is because we are accessing a massive number of rows instead of only a small portion of the table.

Index3: on breed.typical_age

```
-> Nested loop inner join  (cost=1929.88 rows=5411) (actual time=0.320..13.704
rows=5177 loops=1)
    -> Nested loop inner join  (cost=36.20 rows=68) (actual time=0.166..0.529
rows=65 loops=1)
        -> Filter: ((Breed.typical_age > 10) and (Breed.typical_height > (select
#3)))  (cost=12.28 rows=68) (actual time=0.152..0.313 rows=65 loops=1)
            -> Table scan on Breed  (cost=12.28 rows=247) (actual
time=0.034..0.137 rows=247 loops=1)
            -> Select #3 (subquery in condition; run only once)
                -> Aggregate: avg(Breed.typical_height)  (cost=50.65 rows=247)
(actual time=0.106..0.106 rows=1 loops=1)
                    -> Table scan on Breed  (cost=25.95 rows=247) (actual
time=0.018..0.087 rows=247 loops=1)
        -> Single-row index lookup on Breed using PRIMARY (name=Breed.`name`)
(cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=65)
    -> Index lookup on Pet using BREEDNAMEINDEX (breed_name=Breed.`name`)
(cost=19.91 rows=79) (actual time=0.070..0.198 rows=80 loops=65)
```

Similar to the previous indexing, indexing on typical_age does not change the cost.

2. **List the top 15 pet stores that sell dogs with the highest average rating in recent 2 years (to be exact, 365 * 2 days).**

Query

```
SELECT PetShop.*, AVG(rating) AS avg_rating
FROM PetShop JOIN Review ON PetShop.`name` = Review.pet_shop_name
WHERE DATEDIFF(CURRENT_DATE(), Review.review_date) <= 2 * 365
GROUP BY Review.pet_shop_name
ORDER BY avg_rating DESC
LIMIT 15;
```

Output of EXPLAIN ANALYZE (before indexing):

```
-> Limit: 15 row(s)  (actual time=113.207..113.210 rows=15 loops=1)
      -> Sort: avg_rating DESC, limit input to 15 row(s) per chunk  (actual
time=113.207..113.208 rows=15 loops=1)
      -> Table scan on <temporary>  (actual time=0.002..0.283 rows=1000
loops=1)
            -> Aggregate using temporary table  (actual time=112.471..112.819
rows=1000 loops=1)
            -> Nested loop inner join  (cost=16849.82 rows=47850) (actual
time=0.236..98.655 rows=12719 loops=1)
                  -> Table scan on PetShop  (cost=102.25 rows=1000) (actual
time=0.056..0.604 rows=1000 loops=1)
                  -> Filter: ((<cache>(to_days(curdate())) -
to_days(Review.review_date)) <= <cache>((2 * 365)))  (cost=11.97 rows=48)
(actual time=0.065..0.097 rows=13 loops=1000)
                        -> Index lookup on Review using pet_shop_name
(pet_shop_name=PetShop.`name`)  (cost=11.97 rows=48) (actual time=0.064..0.091
rows=50 loops=1000)
```

Index 1: on Review.review_date

If we build an index on review_date, we can easily retrieve the reviews that are within 2 years, which would reduce the size of the Review table before joining. Therefore this is a promising optimization. The index is defined as: `CREATE INDEX reviewDateIdx ON Review (review_date);`

After creating this index, the output of EXPLAIN ANALYSIS is:

```
-> Limit: 15 row(s)  (actual time=113.831..113.834 rows=15 loops=1)
      -> Sort: avg_rating DESC, limit input to 15 row(s) per chunk  (actual
time=113.831..113.833 rows=15 loops=1)
```

```
      -> Table scan on <temporary>  (actual time=0.002..0.272 rows=1000
loops=1)
            -> Aggregate using temporary table  (actual time=113.110..113.437
rows=1000 loops=1)
            -> Nested loop inner join  (cost=18428.64 rows=52361) (actual
time=0.246..98.975 rows=12719 loops=1)
                  -> Table scan on PetShop  (cost=102.25 rows=1000) (actual
time=0.045..0.605 rows=1000 loops=1)
                  -> Filter: ((<cache>(to_days(curdate())) -
to_days(Review.review_date)) <= <cache>((2 * 365)))  (cost=13.10 rows=52)
(actual time=0.064..0.097 rows=13 loops=1000)
                        -> Index lookup on Review using pet_shop_name
(pet_shop_name=PetShop.`name`)  (cost=13.10 rows=52) (actual time=0.063..0.091
rows=50 loops=1000)
```

The new index is not used at all, which is not an expected behavior. If we look into the output of EXPLAIN ANALYSIS, we discover that it first loops for each row in the PetShop table, then finds the reviews on this pet shop using an index, and finally applies the DATEDIFF filter. Therefore the optimizer is not doing what we want: first filter the Review table to get the reviews posted in recent 2 years, then do the joining. This is most likely because after filtering, the primary index on Petshop.name would be invalidated, thus reducing the speed of join.

Index 2: on Review.pet_shop_name
The step that takes the most of the time is the nested loop inner join, which joins PetShop and Review table. Therefore to speed up the query, I tried to add indices on the join attributes. This index is defined as: `CREATE INDEX petShopNameIdx ON Review (pet_shop_name);`

After adding this index, EXPLAIN ANALYSIS gives these outputs:

```
-> Limit: 15 row(s)  (actual time=114.444..114.447 rows=15 loops=1)
      -> Sort: avg_rating DESC, limit input to 15 row(s) per chunk  (actual
time=114.444..114.446 rows=15 loops=1)
      -> Table scan on <temporary>  (actual time=0.002..0.272 rows=1000
loops=1)
            -> Aggregate using temporary table  (actual time=113.683..114.037
rows=1000 loops=1)
            -> Nested loop inner join  (cost=17318.35 rows=49189) (actual
time=0.184..99.319 rows=12719 loops=1)
                  -> Table scan on PetShop  (cost=102.25 rows=1000) (actual
time=0.047..0.681 rows=1000 loops=1)
                  -> Filter: ((<cache>(to_days(curdate())) -
to_days(Review.review_date)) <= <cache>((2 * 365)))  (cost=12.30 rows=49)
(actual time=0.065..0.097 rows=13 loops=1000)
                        -> Index lookup on Review using petShopNameIdx
(pet_shop_name=PetShop.`name`)  (cost=12.30 rows=49) (actual time=0.064..0.091
rows=50 loops=1000)
```

Although petShopIdx is used, there is little time difference with and without the index! It looks like MySQL automatically builds this index to optimize the join operation, even though we do not create it explicitly. Since building an index does not take much time, the final cost is nearly the same.

Index 3: on Review.rating

I tried to build this index, since the rating attribute is used in the query, and building an index might help with the performance. This index is defined as: `CREATE INDEX ratingIdx ON Review (rating);`

The output of EXPLAIN ANALYZE after creating this index is this:

```
-> Limit: 15 row(s)  (actual time=114.639..114.642 rows=15 loops=1)
      -> Sort: avg_rating DESC, limit input to 15 row(s) per chunk  (actual
time=114.639..114.641 rows=15 loops=1)
      -> Table scan on <temporary>  (actual time=0.002..0.285 rows=1000
loops=1)
            -> Aggregate using temporary table  (actual time=113.902..114.246
rows=1000 loops=1)
            -> Nested loop inner join  (cost=17892.80 rows=50830) (actual
time=0.278..99.749 rows=12719 loops=1)
                  -> Table scan on PetShop  (cost=102.25 rows=1000) (actual
time=0.048..0.613 rows=1000 loops=1)
                  -> Filter: ((<cache>(to_days(curdate())) -
to_days(Review.review_date)) <= <cache>((2 * 365)))  (cost=12.71 rows=51)
(actual time=0.066..0.098 rows=13 loops=1000)
                        -> Index lookup on Review using pet_shop_name
(pet_shop_name=PetShop.`name`)  (cost=12.71 rows=51) (actual time=0.065..0.092
rows=50 loops=1000)
```

The new index is not used at all! The output is the same as the case without the index. After further analysis, we find that the joined table is aggregated, then followed by a table scan. After aggregation, our new index is no longer valid and therefore is not used at all.