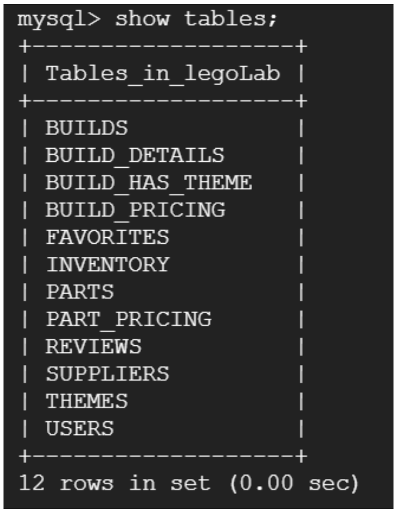
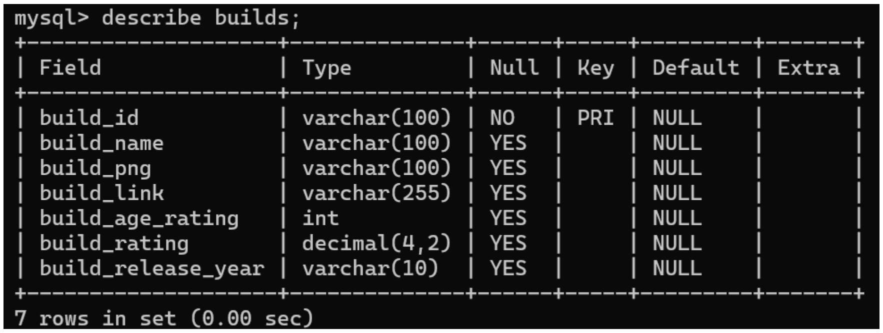
DDL Commands

```
        Create Table USERS (
                user_id VARCHAR(100) Primary key,
                username VARCHAR(100),
                password VARCHAR(255),
                user_email VARCHAR(100),
                user_age INT
                );

        Create table PARTS (
                part_id VARCHAR(100),
                part_color  VARCHAR(100),
                part_name VARCHAR(100),
                part_png VARCHAR(255),
                part_dimensions VARCHAR(255),
                PRIMARY KEY(part_id, part_color)
                );


        Create Table BUILDS (
                build_id VARCHAR(100) Primary Key,
                build_name VARCHAR(100),
                build_png VARCHAR(100),
                build_link VARCHAR(255),
                build_age_rating INTEGER,
                build_rating DECIMAL(4, 2),
                build_release_year VARCHAR(10)
                );

        Create Table THEMES (
                theme_id VARCHAR(100) Primary Key,
                theme_name VARCHAR(100),
                theme_description VARCHAR(1000),
                popular_build_id_1 VARCHAR(100),
                popular_build_id_2 VARCHAR(100),
                popular_build_id_3 VARCHAR(100),
                FOREIGN KEY (popular_build_id_1) REFERENCES
        BUILDS(build_id),
	FOREIGN KEY (popular_build_id_2) REFERENCES BUILDS(build_id),
	FOREIGN KEY (popular_build_id_3) REFERENCES BUILDS(build_id)
	);

Create Table SUPPLIERS (
	supplier_id VARCHAR(100) Primary Key,
	supplier_name VARCHAR(100),
	supplier_region VARCHAR(100)
	);

Create Table INVENTORY (
	user_id VARCHAR(100),
	part_id VARCHAR(100),
	part_color VARCHAR(100),
	part_quantity INTEGER,
	PRIMARY KEY(user_id, part_id, part_color),
	FOREIGN KEY (user_id) REFERENCES USERS(user_id),
	FOREIGN KEY (part_id, part_color) REFERENCES PARTS(part_id, part_color)
	);

Create Table BUILD_DETAILS (
	build_id VARCHAR(100),
	part_id VARCHAR(100),
	part_color VARCHAR(100),
	part_quantity INTEGER,
	PRIMARY KEY(build_id, part_id, part_color),
	FOREIGN KEY (build_id) REFERENCES BUILDS(build_id),
	FOREIGN KEY (part_id, part_color) REFERENCES PARTS(part_id, part_color)
	);

Create Table FAVORITES(
	user_id VARCHAR(100),
	part_id VARCHAR(100),
	part_color VARCHAR(100),
	PRIMARY KEY(user_id, part_id, part_color),
	FOREIGN KEY (user_id) REFERENCES USERS(user_id),
        FOREIGN KEY (part_id, part_color) REFERENCES PARTS(part_id, part_color)
        );

Create Table REVIEWS(
        user_id VARCHAR(100),
        build_id VARCHAR(100),
        review_text VARCHAR(1000),
        PRIMARY KEY(user_id),
        FOREIGN KEY (user_id) REFERENCES USERS(user_id),
        FOREIGN KEY (build_id) REFERENCES BUILDS(build_id)
        );

Create Table BUILD_PRICING (
        supplier_id VARCHAR(100),
        build_id VARCHAR(100),
        PRIMARY KEY(supplier_id, build_id),
        FOREIGN KEY (build_id) REFERENCES BUILDS(build_id),
        FOREIGN KEY (supplier_id) REFERENCES SUPPLIERS(supplier_id),
        build_price DECIMAL(6, 2)
        );

Create Table PART_PRICING (
        supplier_id VARCHAR(100),
        part_id VARCHAR(100),
        part_color VARCHAR(100),
        PRIMARY KEY(supplier_id, part_id, part_color),
        FOREIGN KEY (part_id, part_color) REFERENCES PARTS(part_id, part_color),
        FOREIGN KEY (supplier_id) REFERENCES SUPPLIERS(supplier_id),
        part_price DECIMAL(6, 2)
        );

Create Table BUILD_HAS_THEME (
        theme_id VARCHAR(100),
        build_id VARCHAR(100),
        PRIMARY KEY(theme_id, build_id),
        FOREIGN KEY (theme_id) REFERENCES THEMES(theme_id),
        FOREIGN KEY (build_id) REFERENCES BUILDS(build_id)
        );
```

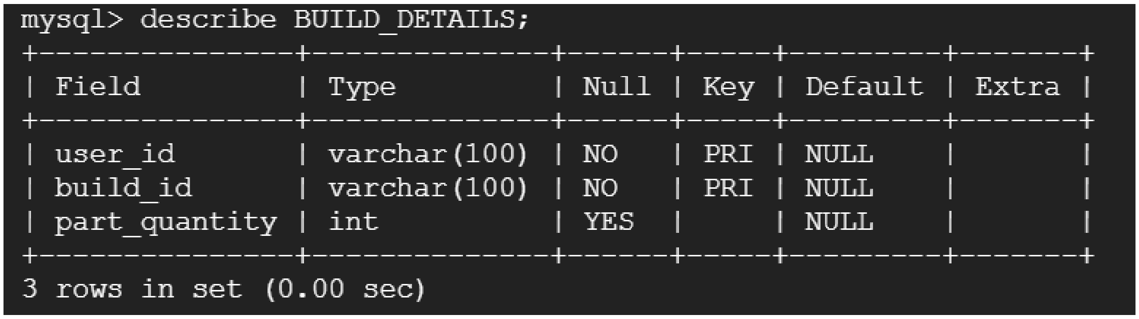
GET the Fields of each Table



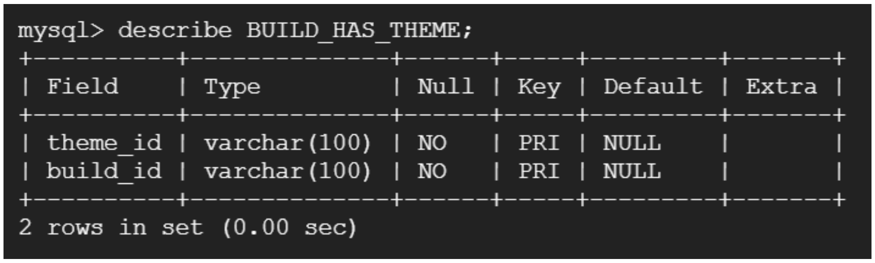
```
mysql> show tables;
+-------------------+
| Tables_in_legoLab |
+-------------------+
| BUILDS            |
| BUILD_DETAILS     |
| BUILD_HAS_THEME   |
| BUILD_PRICING     |
| FAVORITES         |
| INVENTORY         |
| PARTS             |
| PART_PRICING      |
| REVIEWS           |
| SUPPLIERS         |
| THEMES            |
| USERS             |
+-------------------+
12 rows in set (0.00 sec)
```



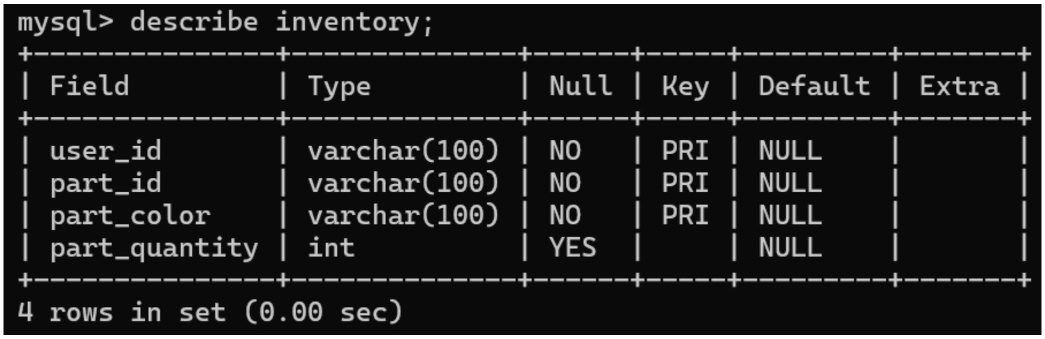
```
mysql> describe builds;
+--------------------+--------------+------+-----+---------+-------+
| Field              | Type         | Null | Key | Default | Extra |
+--------------------+--------------+------+-----+---------+-------+
| build_id           | varchar(100) | NO   | PRI | NULL    |       |
| build_name         | varchar(100) | YES  |     | NULL    |       |
| build_png          | varchar(100) | YES  |     | NULL    |       |
| build_link         | varchar(255) | YES  |     | NULL    |       |
| build_age_rating   | int          | YES  |     | NULL    |       |
| build_rating       | decimal(4,2) | YES  |     | NULL    |       |
| build_release_year | varchar(10)  | YES  |     | NULL    |       |
+--------------------+--------------+------+-----+---------+-------+
7 rows in set (0.00 sec)
```



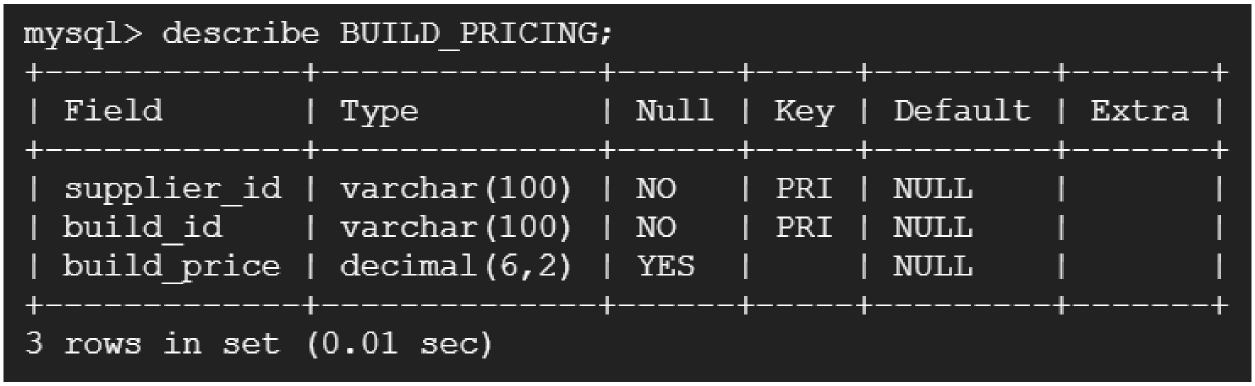
```
mysql> describe BUILD_DETAILS;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| user_id       | varchar(100) | NO   | PRI | NULL    |       |
| build_id      | varchar(100) | NO   | PRI | NULL    |       |
| part_quantity | int          | YES  |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```



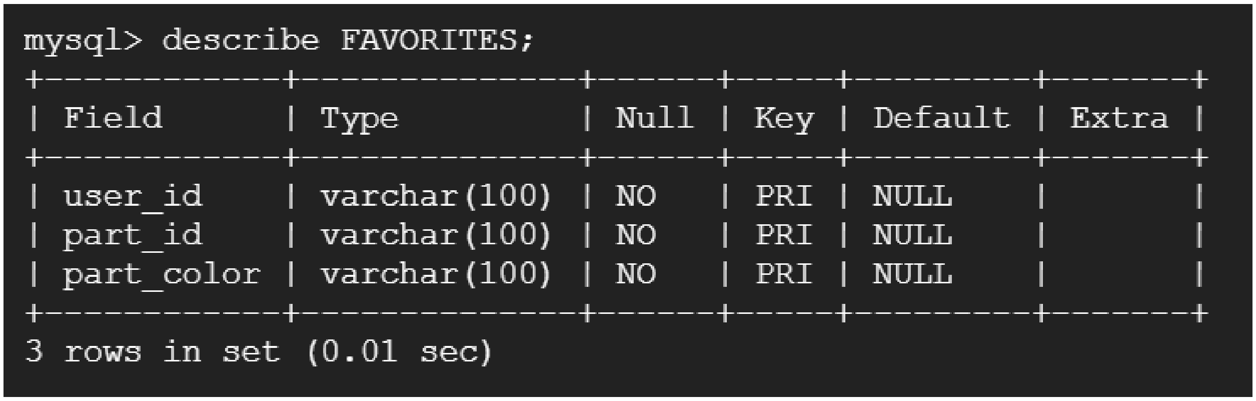
```
mysql> describe BUILD_HAS_THEME;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| theme_id | varchar(100) | NO   | PRI | NULL    |       |
| build_id | varchar(100) | NO   | PRI | NULL    |       |
+----------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```



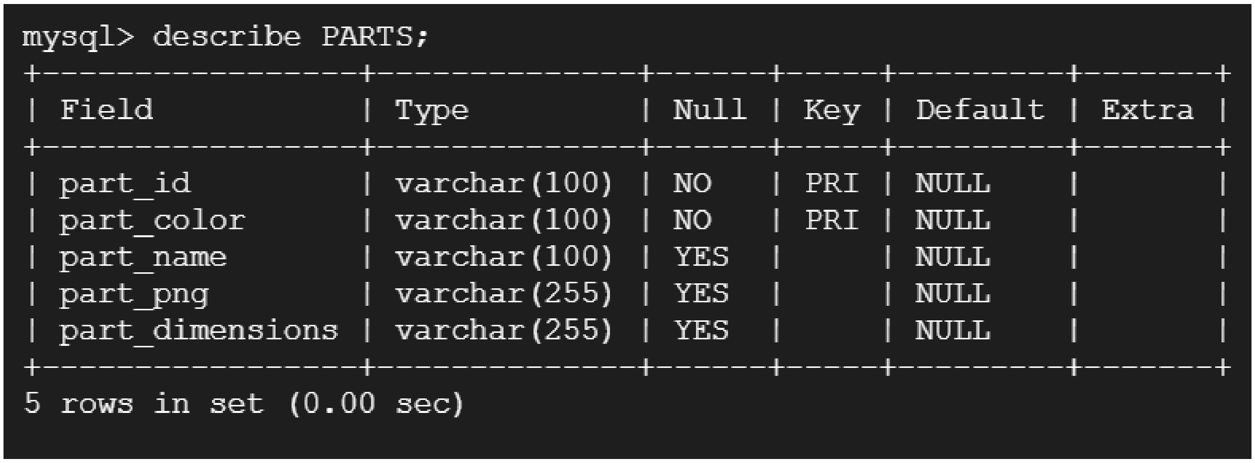
```
mysql> describe inventory;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| user_id       | varchar(100) | NO   | PRI | NULL    |       |
| part_id       | varchar(100) | NO   | PRI | NULL    |       |
| part_color    | varchar(100) | NO   | PRI | NULL    |       |
| part_quantity | int          | YES  |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```



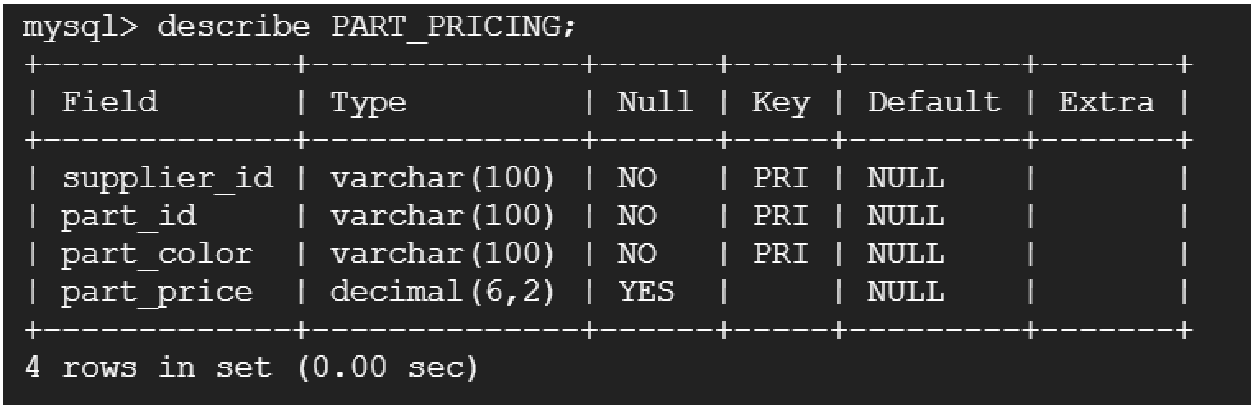
```
mysql> describe BUILD_PRICING;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| supplier_id | varchar(100) | NO   | PRI | NULL    |       |
| build_id    | varchar(100) | NO   | PRI | NULL    |       |
| build_price | decimal(6,2) | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```



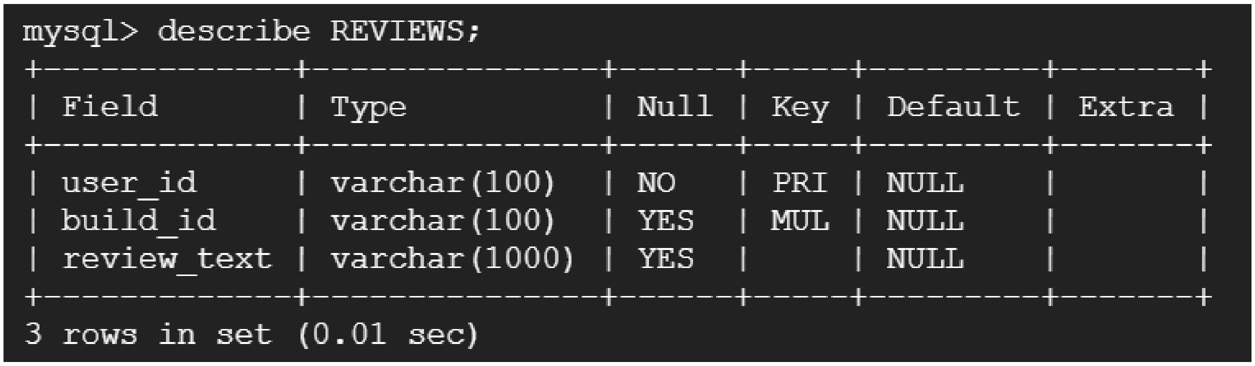
```
mysql> describe FAVORITES;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| user_id    | varchar(100) | NO   | PRI | NULL    |       |
| part_id    | varchar(100) | NO   | PRI | NULL    |       |
| part_color | varchar(100) | NO   | PRI | NULL    |       |
+------------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```



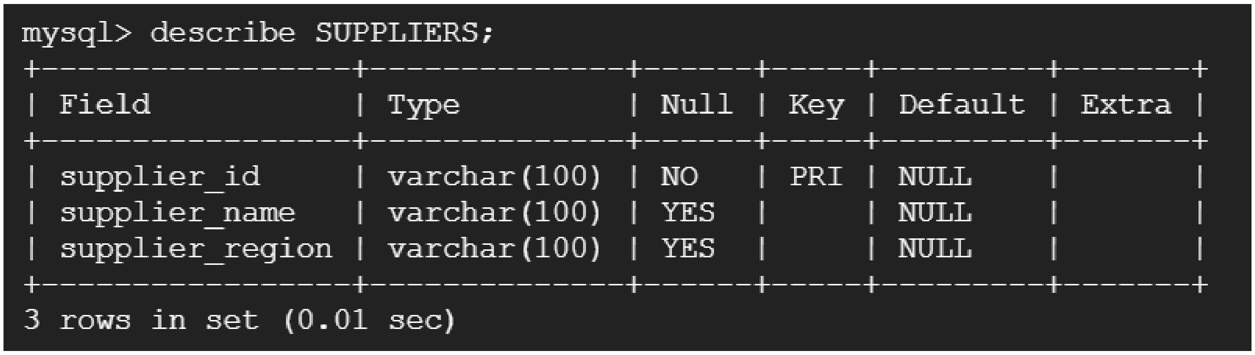
```
mysql> describe PARTS;
+-----------------+--------------+------+-----+---------+-------+
| Field           | Type         | Null | Key | Default | Extra |
+-----------------+--------------+------+-----+---------+-------+
| part_id         | varchar(100) | NO   | PRI | NULL    |       |
| part_color      | varchar(100) | NO   | PRI | NULL    |       |
| part_name       | varchar(100) | YES  |     | NULL    |       |
| part_png        | varchar(255) | YES  |     | NULL    |       |
| part_dimensions | varchar(255) | YES  |     | NULL    |       |
+-----------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```



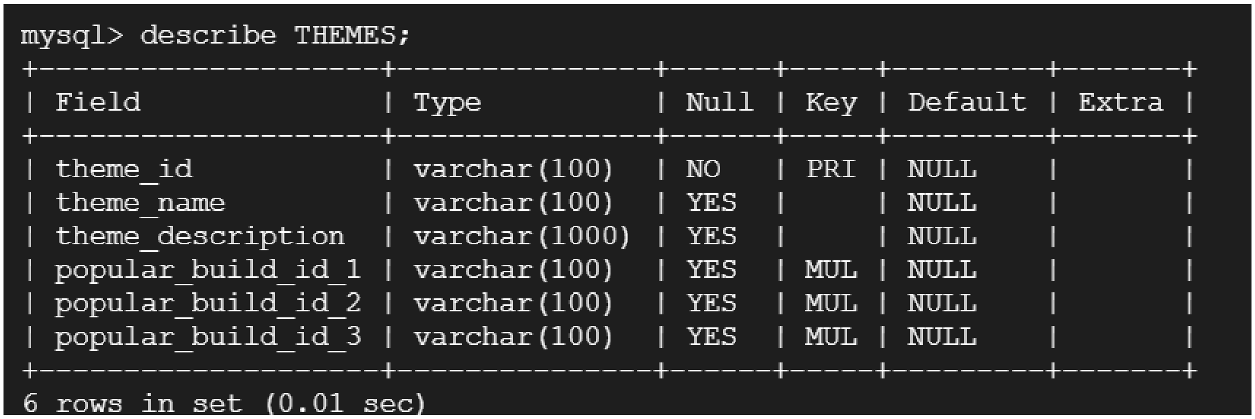
```
mysql> describe PART_PRICING;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| supplier_id | varchar(100) | NO   | PRI | NULL    |       |
| part_id     | varchar(100) | NO   | PRI | NULL    |       |
| part_color  | varchar(100) | NO   | PRI | NULL    |       |
| part_price  | decimal(6,2) | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```



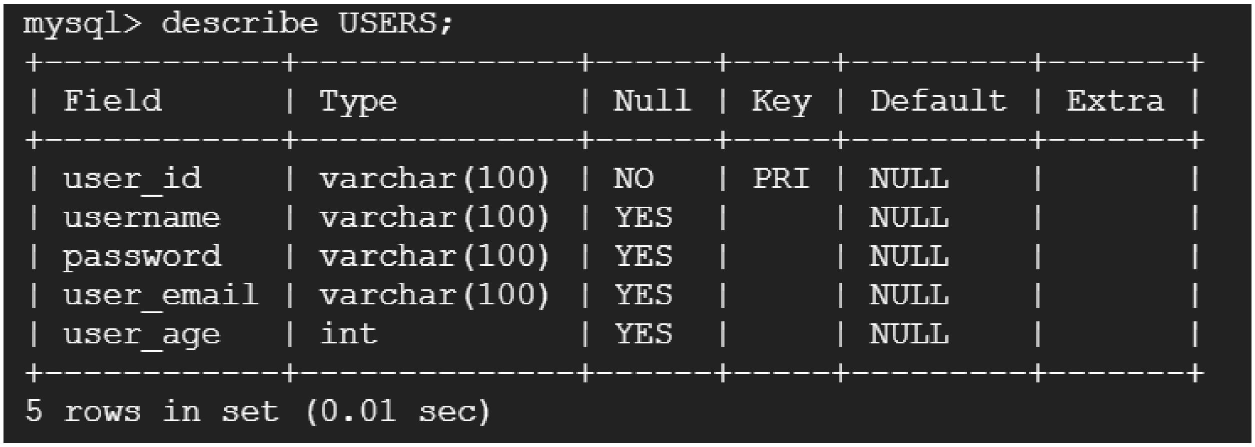
```
mysql> describe REVIEWS;
+-------------+---------------+------+-----+---------+-------+
| Field       | Type          | Null | Key | Default | Extra |
+-------------+---------------+------+-----+---------+-------+
| user_id     | varchar(100)  | NO   | PRI | NULL    |       |
| build_id    | varchar(100)  | YES  | MUL | NULL    |       |
| review_text | varchar(1000) | YES  |     | NULL    |       |
+-------------+---------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```



```
mysql> describe SUPPLIERS;
+-----------------+--------------+------+-----+---------+-------+
| Field           | Type         | Null | Key | Default | Extra |
+-----------------+--------------+------+-----+---------+-------+
| supplier_id     | varchar(100) | NO   | PRI | NULL    |       |
| supplier_name   | varchar(100) | YES  |     | NULL    |       |
| supplier_region | varchar(100) | YES  |     | NULL    |       |
+-----------------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

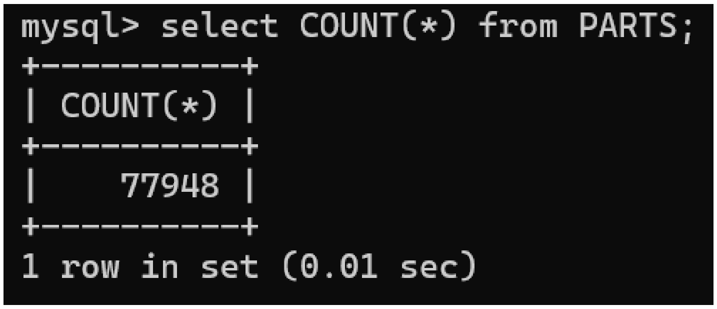


```
mysql> describe THEMES;
+--------------------+---------------+------+-----+---------+-------+
| Field              | Type          | Null | Key | Default | Extra |
+--------------------+---------------+------+-----+---------+-------+
| theme_id           | varchar(100)  | NO   | PRI | NULL    |       |
| theme_name         | varchar(100)  | YES  |     | NULL    |       |
| theme_description  | varchar(1000) | YES  |     | NULL    |       |
| popular_build_id_1 | varchar(100)  | YES  | MUL | NULL    |       |
| popular_build_id_2 | varchar(100)  | YES  | MUL | NULL    |       |
| popular_build_id_3 | varchar(100)  | YES  | MUL | NULL    |       |
+--------------------+---------------+------+-----+---------+-------+
6 rows in set (0.01 sec)
```

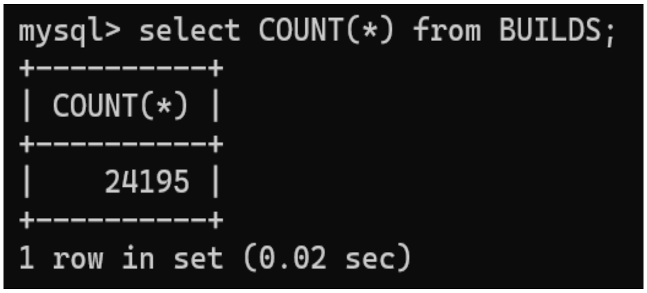


```
mysql> describe USERS;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| user_id    | varchar(100) | NO   | PRI | NULL    |       |
| username   | varchar(100) | YES  |     | NULL    |       |
| password   | varchar(100) | YES  |     | NULL    |       |
| user_email | varchar(100) | YES  |     | NULL    |       |
| user_age   | int          | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

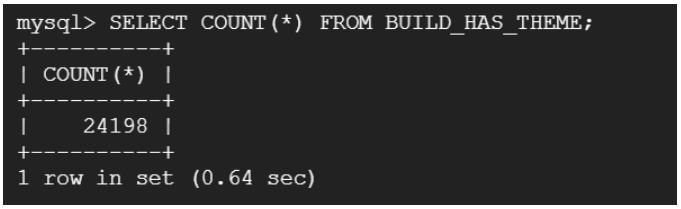
Count of Each Table



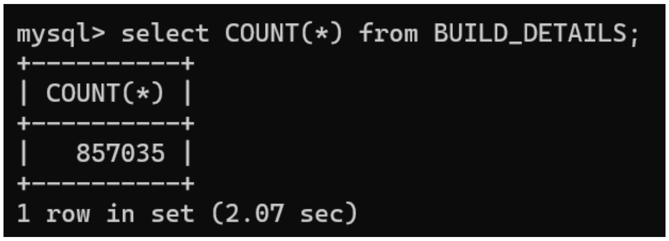
```
mysql> select COUNT(*) from PARTS;
+----------+
| COUNT(*) |
+----------+
|    77948 |
+----------+
1 row in set (0.01 sec)
```



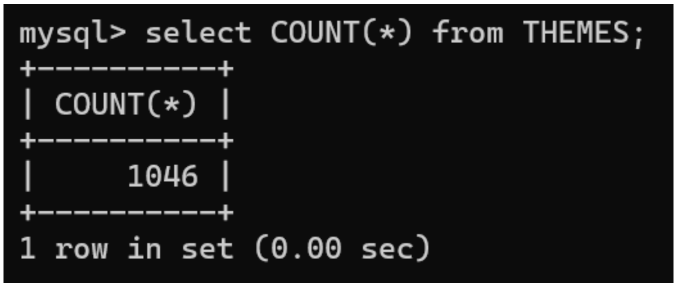
```
mysql> select COUNT(*) from BUILDS;
+----------+
| COUNT(*) |
+----------+
|    24195 |
+----------+
1 row in set (0.02 sec)
```



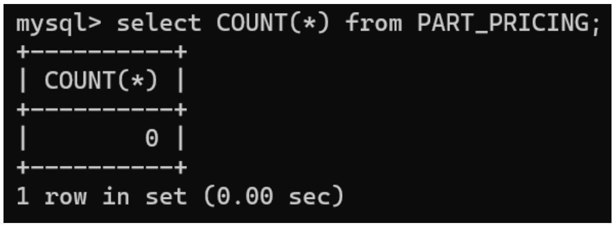
```
mysql> SELECT COUNT(*) FROM BUILD_HAS_THEME;
+----------+
| COUNT(*) |
+----------+
|    24198 |
+----------+
1 row in set (0.64 sec)
```



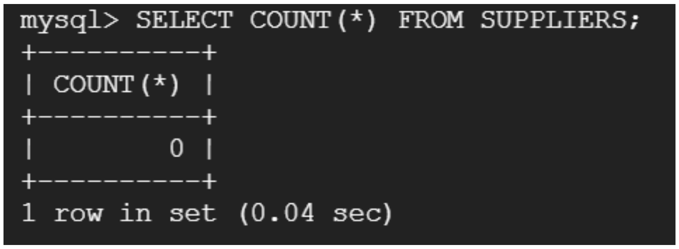
```
mysql> select COUNT(*) from BUILD_DETAILS;
+----------+
| COUNT(*) |
+----------+
|   857035 |
+----------+
1 row in set (2.07 sec)
```



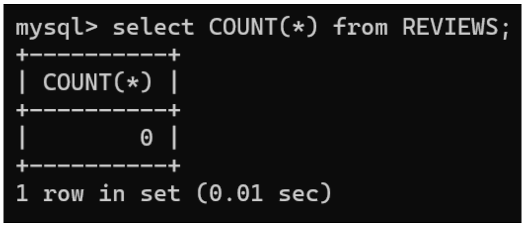
```
mysql> select COUNT(*) from THEMES;
+----------+
| COUNT(*) |
+----------+
|     1046 |
+----------+
1 row in set (0.00 sec)
```



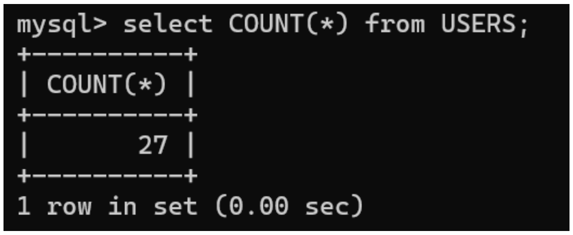
```
mysql> select COUNT(*) from PART_PRICING;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.00 sec)
```



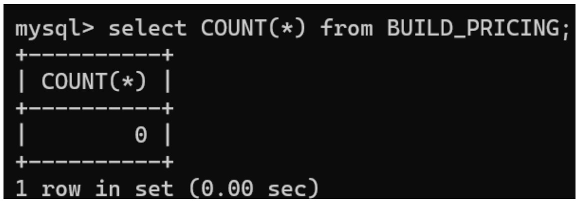
```
mysql> SELECT COUNT(*) FROM SUPPLIERS;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.04 sec)
```



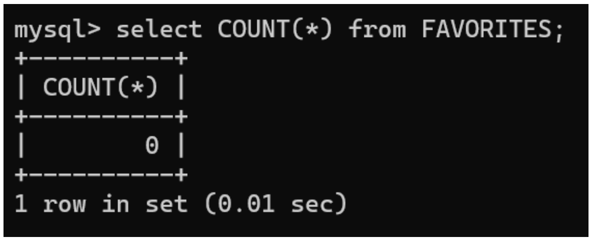
```
mysql> select COUNT(*) from REVIEWS;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.01 sec)
```



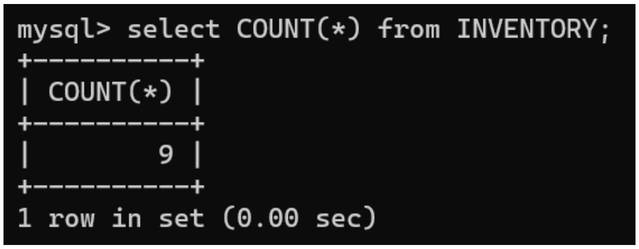
```
mysql> select COUNT(*) from USERS;
+----------+
| COUNT(*) |
+----------+
|       27 |
+----------+
1 row in set (0.00 sec)
```



```
mysql> select COUNT(*) from BUILD_PRICING;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.00 sec)
```



```
mysql> select COUNT(*) from FAVORITES;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.01 sec)
```



```
mysql> select COUNT(*) from INVENTORY;
+----------+
| COUNT(*) |
+----------+
|        9 |
+----------+
1 row in set (0.00 sec)
```

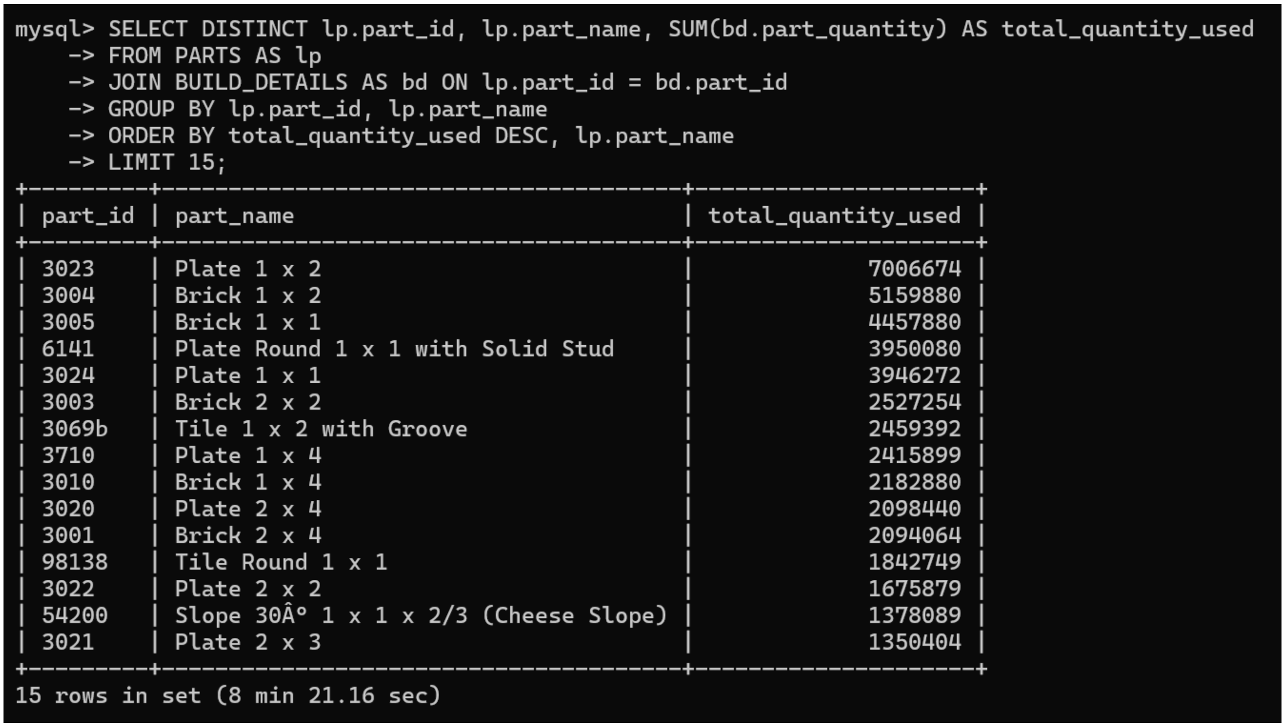
Queries:

1. Query: Find the Most Frequently Used Parts Across All Builds

This query identifies the parts most frequently used across all builds by joining the PARTS and Build_Details tables, aggregating by part_id and part_color.

SQL Concepts: **Join multiple relations, Aggregation with GROUP BY**

```
SELECT DISTINCT lp.part_id, lp.part_name, SUM(bd.part_quantity) AS
total_quantity_used
FROM PARTS AS lp
JOIN BUILD_DETAILS AS bd ON lp.part_id = bd.part_id
GROUP BY lp.part_id, lp.part_name
ORDER BY total_quantity_used DESC, lp.part_name
LIMIT 15;
```



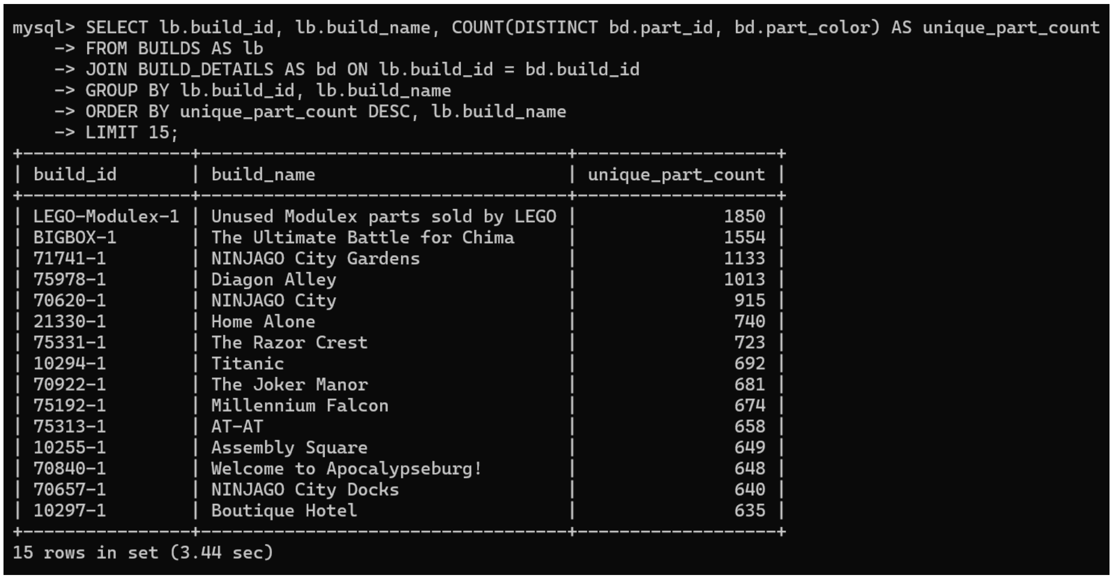
```
mysql> SELECT DISTINCT lp.part_id, lp.part_name, SUM(bd.part_quantity) AS total_quantity_used
    -> FROM PARTS AS lp
    -> JOIN BUILD_DETAILS AS bd ON lp.part_id = bd.part_id
    -> GROUP BY lp.part_id, lp.part_name
    -> ORDER BY total_quantity_used DESC, lp.part_name
    -> LIMIT 15;
+---------+---------------------------------------+---------------------+
| part_id | part_name                             | total_quantity_used |
+---------+---------------------------------------+---------------------+
| 3023    | Plate 1 x 2                           |             7006674 |
| 3004    | Brick 1 x 2                           |             5159880 |
| 3005    | Brick 1 x 1                           |             4457880 |
| 6141    | Plate Round 1 x 1 with Solid Stud     |             3950080 |
| 3024    | Plate 1 x 1                           |             3946272 |
| 3003    | Brick 2 x 2                           |             2527254 |
| 3069b   | Tile 1 x 2 with Groove                |             2459392 |
| 3710    | Plate 1 x 4                           |             2415899 |
| 3010    | Brick 1 x 4                           |             2182880 |
| 3020    | Plate 2 x 4                           |             2098440 |
| 3001    | Brick 2 x 4                           |             2094064 |
| 98138   | Tile Round 1 x 1                      |             1842749 |
| 3022    | Plate 2 x 2                           |             1675879 |
| 54200   | Slope 30Â° 1 x 1 x 2/3 (Cheese Slope) |             1378089 |
| 3021    | Plate 2 x 3                           |             1350404 |
+---------+---------------------------------------+---------------------+
15 rows in set (8 min 21.16 sec)
```

2. Query: Find Builds with the Largest Variety of Unique Parts

This query finds builds with the highest count of unique parts (based on part_id and part_color) used in each build, helping to identify builds that are the most complex in terms of part diversity.

SQL Concepts: **Join multiple relations, Aggregation with GROUP BY**

SELECT lb.build_id, lb.build_name, COUNT(DISTINCT bd.part_id, bd.part_color) AS unique_part_count
FROM BUILDS AS lb
JOIN BUILD_DETAILS AS bd ON lb.build_id = bd.build_id
GROUP BY lb.build_id, lb.build_name
ORDER BY unique_part_count DESC, lb.build_name
LIMIT 15;



```
mysql> SELECT lb.build_id, lb.build_name, COUNT(DISTINCT bd.part_id, bd.part_color) AS unique_part_count
    -> FROM BUILDS AS lb
    -> JOIN BUILD_DETAILS AS bd ON lb.build_id = bd.build_id
    -> GROUP BY lb.build_id, lb.build_name
    -> ORDER BY unique_part_count DESC, lb.build_name
    -> LIMIT 15;
+----------------+-----------------------------------+-------------------+
| build_id       | build_name                        | unique_part_count |
+----------------+-----------------------------------+-------------------+
| LEGO-Modulex-1 | Unused Modulex parts sold by LEGO |              1850 |
| BIGBOX-1       | The Ultimate Battle for Chima     |              1554 |
| 71741-1        | NINJAGO City Gardens              |              1133 |
| 75978-1        | Diagon Alley                      |              1013 |
| 70620-1        | NINJAGO City                      |               915 |
| 21330-1        | Home Alone                        |               740 |
| 75331-1        | The Razor Crest                   |               723 |
| 10294-1        | Titanic                           |               692 |
| 70922-1        | The Joker Manor                   |               681 |
| 75192-1        | Millennium Falcon                 |               674 |
| 75313-1        | AT-AT                             |               658 |
| 10255-1        | Assembly Square                   |               649 |
| 70840-1        | Welcome to Apocalypseburg!        |               648 |
| 70657-1        | NINJAGO City Docks                |               640 |
| 10297-1        | Boutique Hotel                    |               635 |
+----------------+-----------------------------------+-------------------+
15 rows in set (3.44 sec)
```

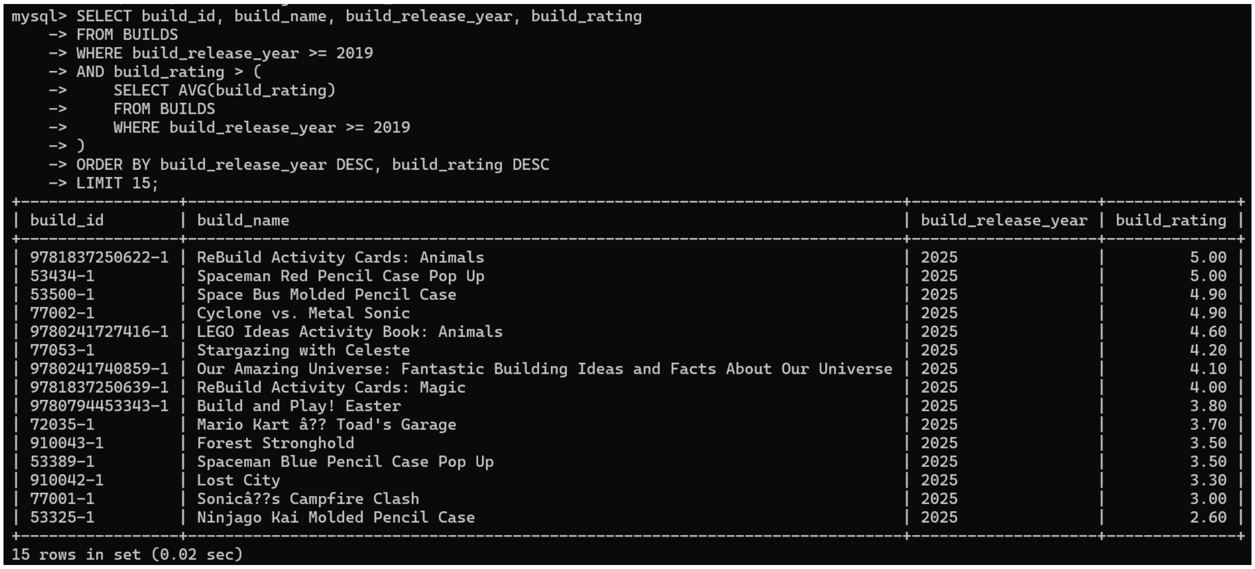
3. Query: Find Builds Released in the Last 5 Years with Above-Average Ratings

This query identifies builds released within the past five years that have a rating above the overall average for that time period.

SQL Concepts: **Subqueries, Aggregation with GROUP BY**

```
SELECT build_id, build_name, build_release_year, build_rating
FROM BUILDS
WHERE build_release_year >= 2019
AND build_rating > (
   SELECT AVG(build_rating)
   FROM BUILDS
   WHERE build_release_year >= 2019
)
```

ORDER BY build_release_year DESC, build_rating DESC
LIMIT 15;



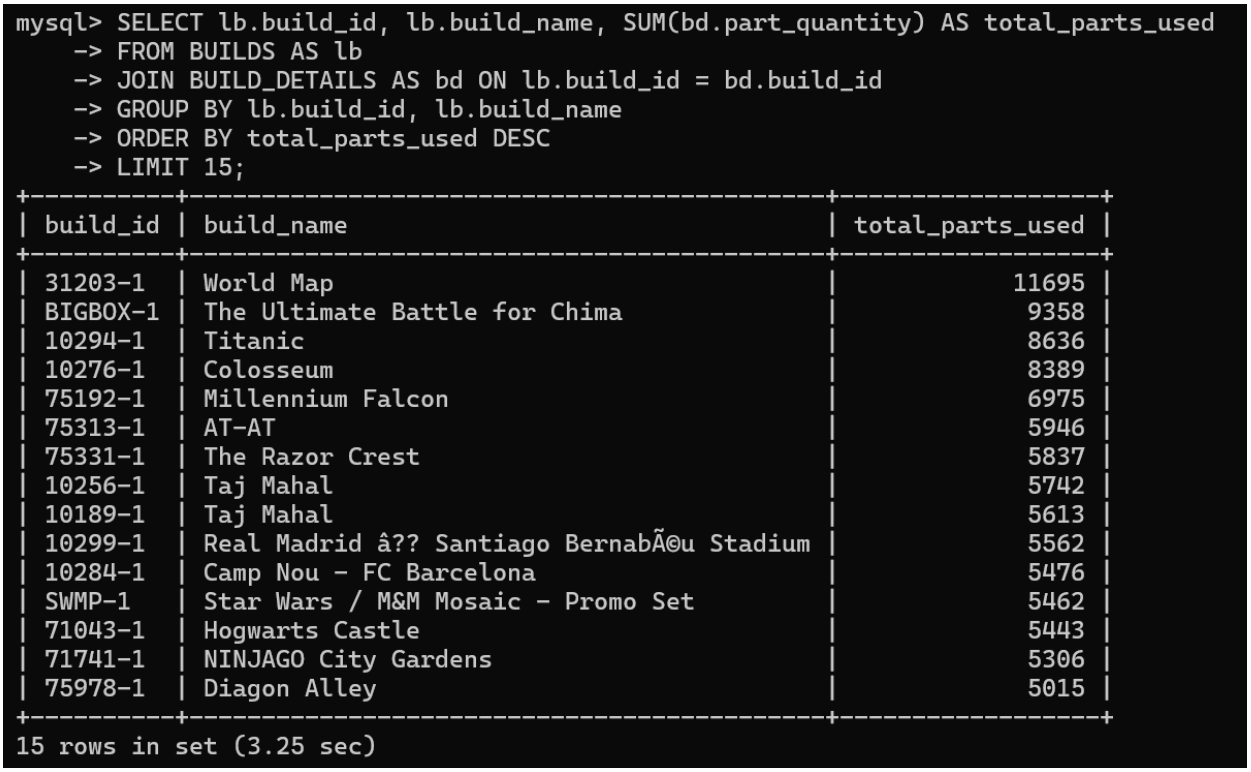
```
mysql> SELECT build_id, build_name, build_release_year, build_rating
    -> FROM BUILDS
    -> WHERE build_release_year >= 2019
    -> AND build_rating > (
    ->     SELECT AVG(build_rating)
    ->     FROM BUILDS
    ->     WHERE build_release_year >= 2019
    -> )
    -> ORDER BY build_release_year DESC, build_rating DESC
    -> LIMIT 15;
+-----------------+-----------------------------------------------------------------------------+--------------------+--------------+
| build_id        | build_name                                                                  | build_release_year | build_rating |
+-----------------+-----------------------------------------------------------------------------+--------------------+--------------+
| 9781837250622-1 | ReBuild Activity Cards: Animals                                             | 2025               |         5.00 |
| 53434-1         | Spaceman Red Pencil Case Pop Up                                             | 2025               |         5.00 |
| 53500-1         | Space Bus Molded Pencil Case                                                | 2025               |         4.90 |
| 77002-1         | Cyclone vs. Metal Sonic                                                     | 2025               |         4.90 |
| 9780241727416-1 | LEGO Ideas Activity Book: Animals                                           | 2025               |         4.60 |
| 77053-1         | Stargazing with Celeste                                                     | 2025               |         4.20 |
| 9780241740859-1 | Our Amazing Universe: Fantastic Building Ideas and Facts About Our Universe | 2025               |         4.10 |
| 9781837250639-1 | ReBuild Activity Cards: Magic                                               | 2025               |         4.00 |
| 9780794453343-1 | Build and Play! Easter                                                      | 2025               |         3.80 |
| 72035-1         | Mario Kart â?? Toad's Garage                                                | 2025               |         3.70 |
| 910043-1        | Forest Stronghold                                                           | 2025               |         3.50 |
| 53389-1         | Spaceman Blue Pencil Case Pop Up                                            | 2025               |         3.50 |
| 910042-1        | Lost City                                                                   | 2025               |         3.30 |
| 77001-1         | Sonicâ??s Campfire Clash                                                    | 2025               |         3.00 |
| 53325-1         | Ninjago Kai Molded Pencil Case                                              | 2025               |         2.60 |
+-----------------+-----------------------------------------------------------------------------+--------------------+--------------+
15 rows in set (0.02 sec)
```

4. Find Builds with the Highest Number of Parts Used

This query lists the builds with the most parts used, based on the sum of part_quantity in Build_Details. This can be useful for identifying complex builds.

SQL Concepts: **Join multiple relations, Aggregation with GROUP BY**

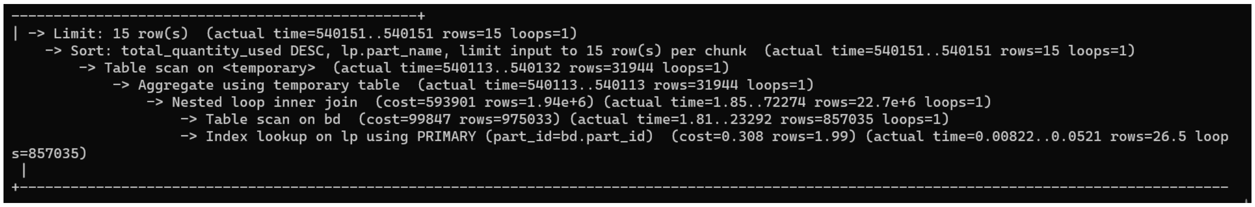
SELECT lb.build_id, lb.build_name, SUM(bd.part_quantity) AS total_parts_used
FROM BUILDS AS lb
JOIN BUILD_DETAILS AS bd ON lb.build_id = bd.build_id
GROUP BY lb.build_id, lb.build_name
ORDER BY total_parts_used DESC
LIMIT 15;



```
mysql> SELECT lb.build_id, lb.build_name, SUM(bd.part_quantity) AS total_parts_used
    -> FROM BUILDS AS lb
    -> JOIN BUILD_DETAILS AS bd ON lb.build_id = bd.build_id
    -> GROUP BY lb.build_id, lb.build_name
    -> ORDER BY total_parts_used DESC
    -> LIMIT 15;
+----------+--------------------------------------------+------------------+
| build_id | build_name                                 | total_parts_used |
+----------+--------------------------------------------+------------------+
| 31203-1  | World Map                                  |            11695 |
| BIGBOX-1 | The Ultimate Battle for Chima              |             9358 |
| 10294-1  | Titanic                                    |             8636 |
| 10276-1  | Colosseum                                  |             8389 |
| 75192-1  | Millennium Falcon                          |             6975 |
| 75313-1  | AT-AT                                      |             5946 |
| 75331-1  | The Razor Crest                            |             5837 |
| 10256-1  | Taj Mahal                                  |             5742 |
| 10189-1  | Taj Mahal                                  |             5613 |
| 10299-1  | Real Madrid â?? Santiago BernabÃ©u Stadium |             5562 |
| 10284-1  | Camp Nou - FC Barcelona                    |             5476 |
| SWMP-1   | Star Wars / M&M Mosaic - Promo Set         |             5462 |
| 71043-1  | Hogwarts Castle                            |             5443 |
| 71741-1  | NINJAGO City Gardens                       |             5306 |
| 75978-1  | Diagon Alley                               |             5015 |
+----------+--------------------------------------------+------------------+
15 rows in set (3.25 sec)
```

INDEXING:
Query 1
Default Index

- Cost: 99847 for table scan
- Time: 23292 on table scan

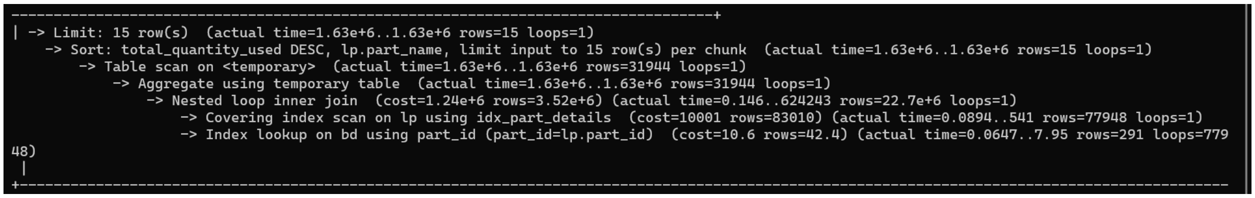


```
------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=540151..540151 rows=15 loops=1)
    -> Sort: total_quantity_used DESC, lp.part_name, limit input to 15 row(s) per chunk  (actual time=540151..540151 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=540113..540132 rows=31944 loops=1)
            -> Aggregate using temporary table  (actual time=540113..540113 rows=31944 loops=1)
                -> Nested loop inner join  (cost=593901 rows=1.94e+6) (actual time=1.85..72274 rows=22.7e+6 loops=1)
                    -> Table scan on bd  (cost=99847 rows=975033) (actual time=1.81..23292 rows=857035 loops=1)
                    -> Index lookup on lp using PRIMARY (part_id=bd.part_id)  (cost=0.308 rows=1.99) (actual time=0.00822..0.0521 rows=26.5 loop
s=857035)
 |
+---------------------------------------------------------------------------------------------------------------------------------------------
```

Index 1:
Index on Parts.part_name

Why: This index will enhance the performance of queries that join the Parts table with the BUILD_DETAILS table, especially when retrieving part quantities.

CREATE INDEX idx_part_details ON PARTS (part_name);



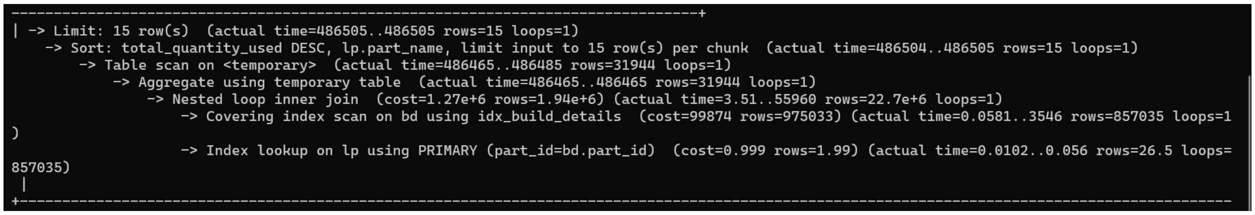
```
---------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=1.63e+6..1.63e+6 rows=15 loops=1)
    -> Sort: total_quantity_used DESC, lp.part_name, limit input to 15 row(s) per chunk  (actual time=1.63e+6..1.63e+6 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=1.63e+6..1.63e+6 rows=31944 loops=1)
            -> Aggregate using temporary table  (actual time=1.63e+6..1.63e+6 rows=31944 loops=1)
                -> Nested loop inner join  (cost=1.24e+6 rows=3.52e+6) (actual time=0.146..624243 rows=22.7e+6 loops=1)
                    -> Covering index scan on lp using idx_part_details  (cost=10001 rows=83010) (actual time=0.0894..541 rows=77948 loops=1)
                    -> Index lookup on bd using part_id (part_id=lp.part_id)  (cost=10.6 rows=42.4) (actual time=0.0647..7.95 rows=291 loops=779
48)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------
```

Findings and Explanation: The actual execution time decreased from 23292 seconds to 541 seconds, showing a measurable improvement. The query cost decreased considerably to 10001 from 99847. The indexing strategy effectively optimized the query by improving the speed of joins, aggregations, and sorting operations, leading to faster execution times and more efficient resource usage.

Index 2:
Index on Build_Details.part_quantity

Why: This index will enhance the performance of queries that join the Build_Details table with the BUILDS table, especially when counting parts quantities across builds

CREATE INDEX idx_build_details ON Build_Details (part_quantity);



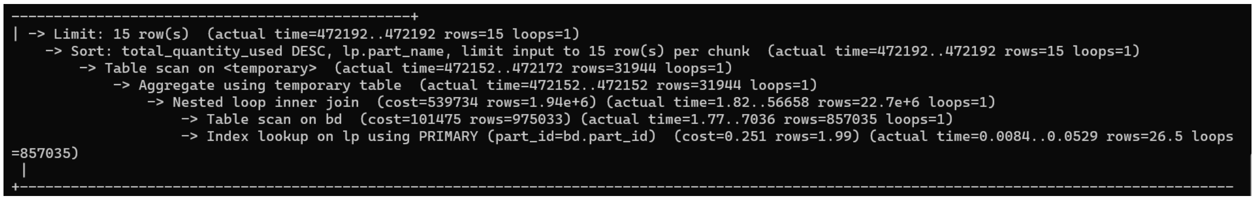
```
---------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=486505..486505 rows=15 loops=1)
    -> Sort: total_quantity_used DESC, lp.part_name, limit input to 15 row(s) per chunk  (actual time=486504..486505 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=486465..486485 rows=31944 loops=1)
            -> Aggregate using temporary table  (actual time=486465..486465 rows=31944 loops=1)
                -> Nested loop inner join  (cost=1.27e+6 rows=1.94e+6) (actual time=3.51..55960 rows=22.7e+6 loops=1)
                    -> Covering index scan on bd using idx_build_details  (cost=99874 rows=975033) (actual time=0.0581..3546 rows=857035 loops=1
)

                    -> Index lookup on lp using PRIMARY (part_id=bd.part_id)  (cost=0.999 rows=1.99) (actual time=0.0102..0.056 rows=26.5 loops=
857035)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------
```

Findings and Explanation: The actual execution time increased from 23292 seconds to 3546 seconds, showing a measurable improvement. The query cost remained roughly the same.The indexing strategy effectively optimized the query by improving the speed of joins, aggregations, and sorting operations, leading to faster execution times and more efficient resource usage.

Index 3:
 Index on Parts.part_name and Parts.part_dimensions

Why: This composite index will enhance the performance of queries that join the Build_Details table with the Part table, especially when retrieving part quantities and parts dimensions builds

CREATE INDEX idx_part_dim_col ON PARTS(part_name, part_dimensions);



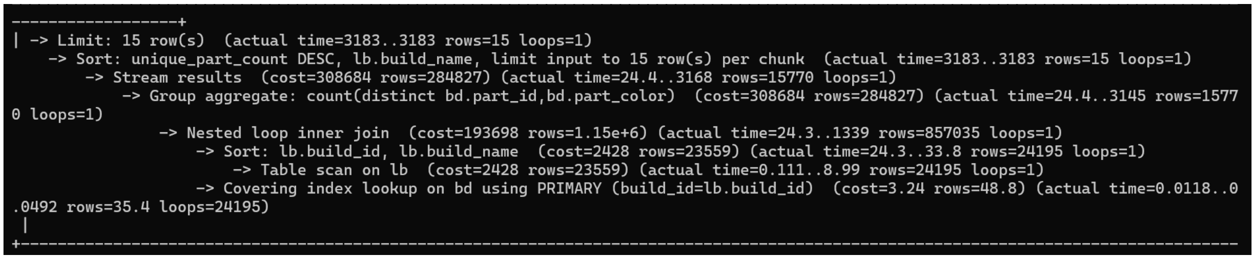
```
-----------------------------------------------+
| -> Limit: 15 row(s)  (actual time=472192..472192 rows=15 loops=1)
    -> Sort: total_quantity_used DESC, lp.part_name, limit input to 15 row(s) per chunk  (actual time=472192..472192 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=472152..472172 rows=31944 loops=1)
            -> Aggregate using temporary table  (actual time=472152..472152 rows=31944 loops=1)
                -> Nested loop inner join  (cost=539734 rows=1.94e+6) (actual time=1.82..56658 rows=22.7e+6 loops=1)
                    -> Table scan on bd  (cost=101475 rows=975033) (actual time=1.77..7036 rows=857035 loops=1)
                    -> Index lookup on lp using PRIMARY (part_id=bd.part_id)  (cost=0.251 rows=1.99) (actual time=0.0084..0.0529 rows=26.5 loops
=857035)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------
```

Findings and Explanation: The actual execution time decreased from 23292 seconds to 7036 seconds, showing a measurable improvement. The query cost increased from 98860 to 99847. The Indexing strategy did not cause a significant improvement.

Query 2

Default Index

- Cost: 2428 on scan
- Time: 3183 on limit



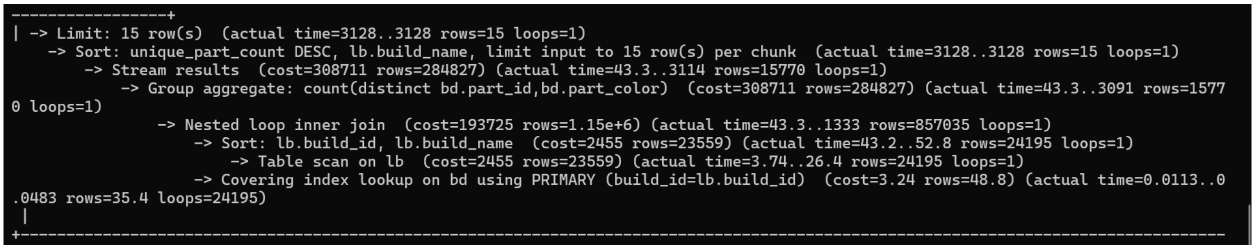
```
------------------+
| -> Limit: 15 row(s)  (actual time=3183..3183 rows=15 loops=1)
    -> Sort: unique_part_count DESC, lb.build_name, limit input to 15 row(s) per chunk  (actual time=3183..3183 rows=15 loops=1)
        -> Stream results  (cost=308684 rows=284827) (actual time=24.4..3168 rows=15770 loops=1)
            -> Group aggregate: count(distinct bd.part_id,bd.part_color)  (cost=308684 rows=284827) (actual time=24.4..3145 rows=1577
0 loops=1)
                -> Nested loop inner join  (cost=193698 rows=1.15e+6) (actual time=24.3..1339 rows=857035 loops=1)
                    -> Sort: lb.build_id, lb.build_name  (cost=2428 rows=23559) (actual time=24.3..33.8 rows=24195 loops=1)
                        -> Table scan on lb  (cost=2428 rows=23559) (actual time=0.111..8.99 rows=24195 loops=1)
                    -> Covering index lookup on bd using PRIMARY (build_id=lb.build_id)  (cost=3.24 rows=48.8) (actual time=0.0118..0
.0492 rows=35.4 loops=24195)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------
```

Index 1:

Index on Build_Details.part_quantity

Why: This index will enhance the performance of queries that join the Build_Details table with the BUILDS table, especially when retrieving part quantities.

CREATE INDEX idx_part_details ON Build_Details (part_quantity);



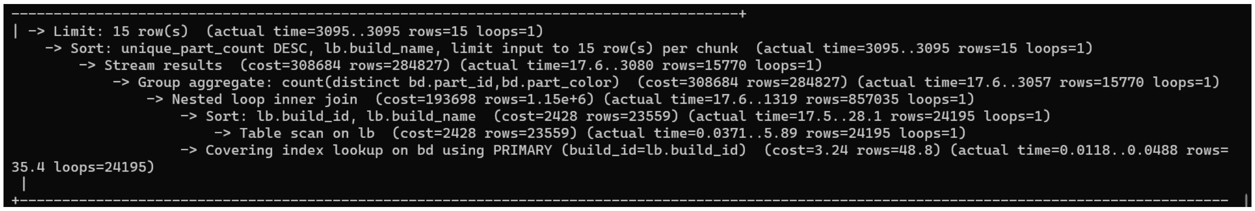
```
-----------------+
| -> Limit: 15 row(s)  (actual time=3128..3128 rows=15 loops=1)
    -> Sort: unique_part_count DESC, lb.build_name, limit input to 15 row(s) per chunk  (actual time=3128..3128 rows=15 loops=1)
        -> Stream results  (cost=308711 rows=284827) (actual time=43.3..3114 rows=15770 loops=1)
            -> Group aggregate: count(distinct bd.part_id,bd.part_color)  (cost=308711 rows=284827) (actual time=43.3..3091 rows=1577
0 loops=1)
                -> Nested loop inner join  (cost=193725 rows=1.15e+6) (actual time=43.3..1333 rows=857035 loops=1)
                    -> Sort: lb.build_id, lb.build_name  (cost=2455 rows=23559) (actual time=43.2..52.8 rows=24195 loops=1)
                        -> Table scan on lb  (cost=2455 rows=23559) (actual time=3.74..26.4 rows=24195 loops=1)
                    -> Covering index lookup on bd using PRIMARY (build_id=lb.build_id)  (cost=3.24 rows=48.8) (actual time=0.0113..0
.0483 rows=35.4 loops=24195)
 |
+------------------------------------------------------------------------------------------------------------------------------------------
```

Findings and Explanation: The actual execution time decreased from 3183 seconds to 26.4 seconds, showing a measurable improvement. The query cost remained roughly the same. The indexing strategy effectively optimized the query by improving the speed of joins, aggregations, and sorting operations, leading to faster execution times and more efficient resource usage.

Index 2:
Index on Builds.release_year

Why: This index will enhance the performance of queries that join the Build_Details table with the BUILDS table, especially when counting unique release across builds

CREATE INDEX idx_build_year ON BUILDS (build_release_year);



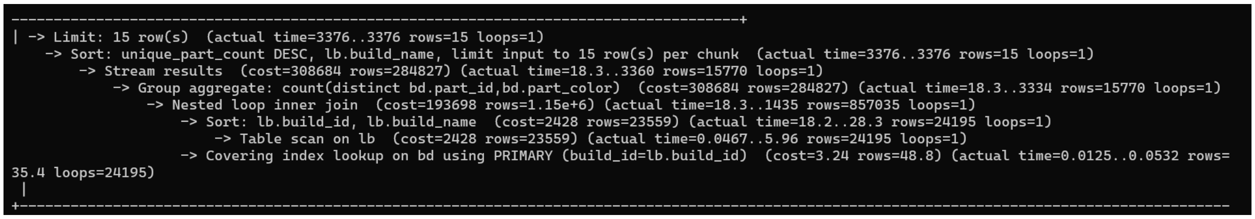
```
-------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=3095..3095 rows=15 loops=1)
    -> Sort: unique_part_count DESC, lb.build_name, limit input to 15 row(s) per chunk  (actual time=3095..3095 rows=15 loops=1)
        -> Stream results  (cost=308684 rows=284827) (actual time=17.6..3080 rows=15770 loops=1)
            -> Group aggregate: count(distinct bd.part_id,bd.part_color)  (cost=308684 rows=284827) (actual time=17.6..3057 rows=15770 loops=1)
                -> Nested loop inner join  (cost=193698 rows=1.15e+6) (actual time=17.6..1319 rows=857035 loops=1)
                    -> Sort: lb.build_id, lb.build_name  (cost=2428 rows=23559) (actual time=17.5..28.1 rows=24195 loops=1)
                        -> Table scan on lb  (cost=2428 rows=23559) (actual time=0.0371..5.89 rows=24195 loops=1)
                    -> Covering index lookup on bd using PRIMARY (build_id=lb.build_id)  (cost=3.24 rows=48.8) (actual time=0.0118..0.0488 rows=
35.4 loops=24195)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------  |
```

Findings and Explanation: The actual execution time decreased from 3183 seconds to 5.89 seconds, showing a measurable improvement. The query cost remained roughly the same. The indexing strategy effectively optimized the query by improving the speed of joins, aggregations, and sorting operations, leading to faster execution times and more efficient resource usage.

Index 3:
Index on BUILDS.age_rating

Why: This composite index will enhance the performance of queries that join the Build_Details table with the BUILDS table, especially when retrieving part quantities and counting unique parts across builds

CREATE INDEX idx_build_age ON BUILDS(build_age_rating);



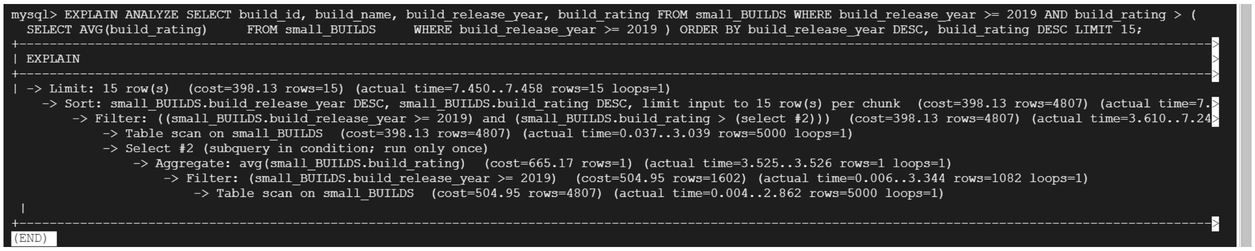
```
-------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=3376..3376 rows=15 loops=1)
    -> Sort: unique_part_count DESC, lb.build_name, limit input to 15 row(s) per chunk  (actual time=3376..3376 rows=15 loops=1)
        -> Stream results  (cost=308684 rows=284827) (actual time=18.3..3360 rows=15770 loops=1)
            -> Group aggregate: count(distinct bd.part_id,bd.part_color)  (cost=308684 rows=284827) (actual time=18.3..3334 rows=15770 loops=1)
                -> Nested loop inner join  (cost=193698 rows=1.15e+6) (actual time=18.3..1435 rows=857035 loops=1)
                    -> Sort: lb.build_id, lb.build_name  (cost=2428 rows=23559) (actual time=18.2..28.3 rows=24195 loops=1)
                        -> Table scan on lb  (cost=2428 rows=23559) (actual time=0.0467..5.96 rows=24195 loops=1)
                    -> Covering index lookup on bd using PRIMARY (build_id=lb.build_id)  (cost=3.24 rows=48.8) (actual time=0.0125..0.0532 rows=
35.4 loops=24195)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------
```

Findings and Explanation: The actual execution time decreased from 3183 seconds to 5.96 seconds, showing a measurable improvement. The query cost remained roughly the same. The indexing strategy effectively optimized the query by improving the speed of joins, aggregations, and sorting operations, leading to faster execution times and more efficient resource usage.

Query 3

Default Index

- Cost: 398.13 on limit
- Time: 7.458 on limit



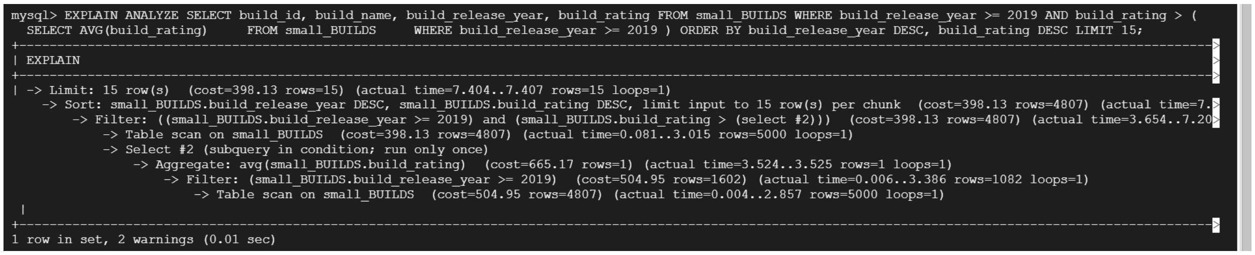
```
mysql> EXPLAIN ANALYZE SELECT build_id, build_name, build_release_year, build_rating FROM small_BUILDS WHERE build_release_year >= 2019 AND build_rating > (
  SELECT AVG(build_rating)     FROM small_BUILDS     WHERE build_release_year >= 2019 ) ORDER BY build_release_year DESC, build_rating DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------------------------------->
| EXPLAIN                                                                                                                                                  >
+--------------------------------------------------------------------------------------------------------------------------------------------------------->
| -> Limit: 15 row(s)  (cost=398.13 rows=15) (actual time=7.450..7.458 rows=15 loops=1)
    -> Sort: small_BUILDS.build_release_year DESC, small_BUILDS.build_rating DESC, limit input to 15 row(s) per chunk  (cost=398.13 rows=4807) (actual time=7.>
        -> Filter: ((small_BUILDS.build_release_year >= 2019) and (small_BUILDS.build_rating > (select #2)))  (cost=398.13 rows=4807) (actual time=3.610..7.24>
            -> Table scan on small_BUILDS  (cost=398.13 rows=4807) (actual time=0.037..3.039 rows=5000 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Aggregate: avg(small_BUILDS.build_rating)  (cost=665.17 rows=1) (actual time=3.525..3.526 rows=1 loops=1)
                    -> Filter: (small_BUILDS.build_release_year >= 2019)  (cost=504.95 rows=1602) (actual time=0.006..3.344 rows=1082 loops=1)
                        -> Table scan on small_BUILDS  (cost=504.95 rows=4807) (actual time=0.004..2.862 rows=5000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------->
(END)
```

Index 1:

Index on Builds.builds_release_year

Why: This index will help improve the speed of the query when we are checking by indexing through build release_year.

CREATE INDEX idx_build_year ON small_BUILDS(build_release_year);



```
mysql> EXPLAIN ANALYZE SELECT build_id, build_name, build_release_year, build_rating FROM small_BUILDS WHERE build_release_year >= 2019 AND build_rating > (
  SELECT AVG(build_rating)     FROM small_BUILDS     WHERE build_release_year >= 2019 ) ORDER BY build_release_year DESC, build_rating DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------------------------------->
| EXPLAIN                                                                                                                                                  >
+--------------------------------------------------------------------------------------------------------------------------------------------------------->
| -> Limit: 15 row(s)  (cost=398.13 rows=15) (actual time=7.404..7.407 rows=15 loops=1)
    -> Sort: small_BUILDS.build_release_year DESC, small_BUILDS.build_rating DESC, limit input to 15 row(s) per chunk  (cost=398.13 rows=4807) (actual time=7.>
        -> Filter: ((small_BUILDS.build_release_year >= 2019) and (small_BUILDS.build_rating > (select #2)))  (cost=398.13 rows=4807) (actual time=3.654..7.20>
            -> Table scan on small_BUILDS  (cost=398.13 rows=4807) (actual time=0.081..3.015 rows=5000 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Aggregate: avg(small_BUILDS.build_rating)  (cost=665.17 rows=1) (actual time=3.524..3.525 rows=1 loops=1)
                    -> Filter: (small_BUILDS.build_release_year >= 2019)  (cost=504.95 rows=1602) (actual time=0.006..3.386 rows=1082 loops=1)
                        -> Table scan on small_BUILDS  (cost=504.95 rows=4807) (actual time=0.004..2.857 rows=5000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------->
1 row in set, 2 warnings (0.01 sec)
```

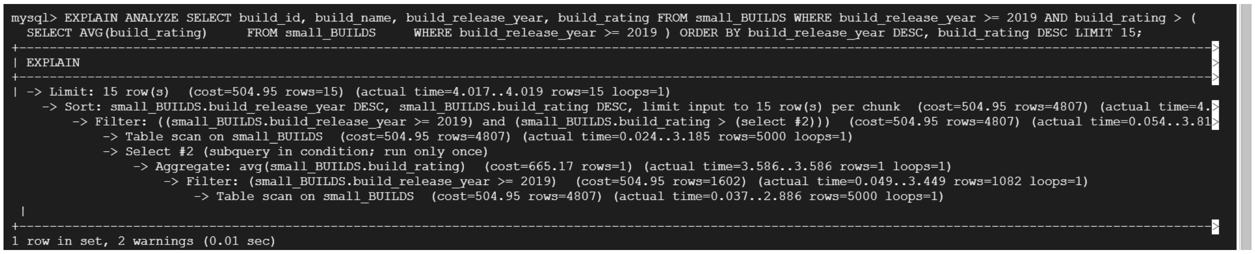
Findings and Explanation: The actual execution time did not decrease at all. The query cost remained roughly the same. The Indexing strategy did not cause a significant improvement.

Index 2:

Index on BUILDS.build_rating

Why: An index on these columns will speed up queries that filter on build_rating, like the one that identifies builds released above a certain rating.

CREATE INDEX idx_build_rating ON BUILDS(build_rating);



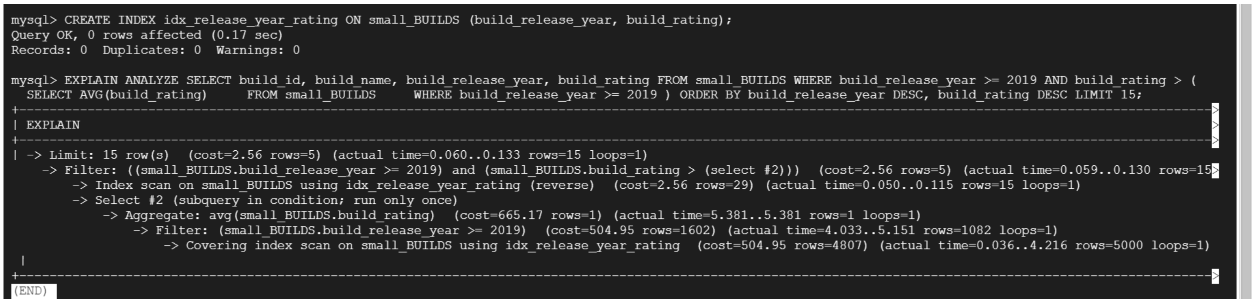
```
mysql> EXPLAIN ANALYZE SELECT build_id, build_name, build_release_year, build_rating FROM small_BUILDS WHERE build_release_year >= 2019 AND build_rating > (
  SELECT AVG(build_rating)     FROM small_BUILDS     WHERE build_release_year >= 2019 ) ORDER BY build_release_year DESC, build_rating DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------------------------------------->
| EXPLAIN                                                                                                                                                         >
+--------------------------------------------------------------------------------------------------------------------------------------------------------------->
| -> Limit: 15 row(s)  (cost=504.95 rows=15) (actual time=4.017..4.019 rows=15 loops=1)
    -> Sort: small_BUILDS.build_release_year DESC, small_BUILDS.build_rating DESC, limit input to 15 row(s) per chunk  (cost=504.95 rows=4807) (actual time=4.>
        -> Filter: ((small_BUILDS.build_release_year >= 2019) and (small_BUILDS.build_rating > (select #2)))  (cost=504.95 rows=4807) (actual time=0.054..3.81>
            -> Table scan on small_BUILDS  (cost=504.95 rows=4807) (actual time=0.024..3.185 rows=5000 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Aggregate: avg(small_BUILDS.build_rating)  (cost=665.17 rows=1) (actual time=3.586..3.586 rows=1 loops=1)
                    -> Filter: (small_BUILDS.build_release_year >= 2019)  (cost=504.95 rows=1602) (actual time=0.049..3.449 rows=1082 loops=1)
                        -> Table scan on small_BUILDS  (cost=504.95 rows=4807) (actual time=0.037..2.886 rows=5000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------->
1 row in set, 2 warnings (0.01 sec)
```

Findings and Explanation: The actual execution time remained the same. The query cost actually increased. The indexing strategy reduced the speed of the query by decreasing the speed of joins, aggregations, and sorting operations, leading to slower execution times and less efficient resource usage.

Index 3:
 Index on BUILDS.builds_release_year and BUILDS.builds_rating

Why: Since the Build_Themes table is often joined with the BUILDS table, indexing theme_id will speed up join operations when retrieving theme information related to builds.

CREATE INDEX idx_release_year_rating ON BUILDS (build_release_year, build_rating);

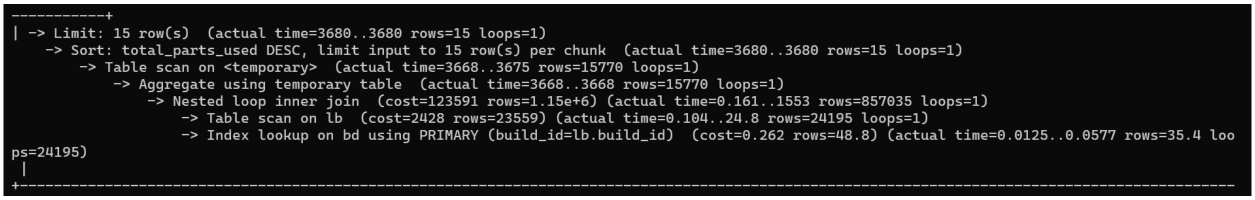


```
mysql> CREATE INDEX idx_release_year_rating ON small_BUILDS (build_release_year, build_rating);
Query OK, 0 rows affected (0.17 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT build_id, build_name, build_release_year, build_rating FROM small_BUILDS WHERE build_release_year >= 2019 AND build_rating > (
  SELECT AVG(build_rating)     FROM small_BUILDS     WHERE build_release_year >= 2019 ) ORDER BY build_release_year DESC, build_rating DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------------------------------------->
| EXPLAIN                                                                                                                                                         >
+--------------------------------------------------------------------------------------------------------------------------------------------------------------->
| -> Limit: 15 row(s)  (cost=2.56 rows=5) (actual time=0.060..0.133 rows=15 loops=1)
    -> Filter: ((small_BUILDS.build_release_year >= 2019) and (small_BUILDS.build_rating > (select #2)))  (cost=2.56 rows=5) (actual time=0.059..0.130 rows=15>
        -> Index scan on small_BUILDS using idx_release_year_rating (reverse)  (cost=2.56 rows=29) (actual time=0.050..0.115 rows=15 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(small_BUILDS.build_rating)  (cost=665.17 rows=1) (actual time=5.381..5.381 rows=1 loops=1)
                -> Filter: (small_BUILDS.build_release_year >= 2019)  (cost=504.95 rows=1602) (actual time=4.033..5.151 rows=1082 loops=1)
                    -> Covering index scan on small_BUILDS using idx_release_year_rating  (cost=504.95 rows=4807) (actual time=0.036..4.216 rows=5000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------->
(END)
```

Findings and Explanation: The actual execution time decreased greatly from 7.45 seconds to 0.06 seconds, showing a measurable improvement. The query cost also decreased considerably to 2.56 from 398.13. The indexing strategy effectively optimized the query by improving the speed of joins, aggregations, and sorting operations, leading to faster execution times and more efficient resource usage.

Query 4
Default Index

- Cost: 2428 on table scan
- Time: 24.8 on limit

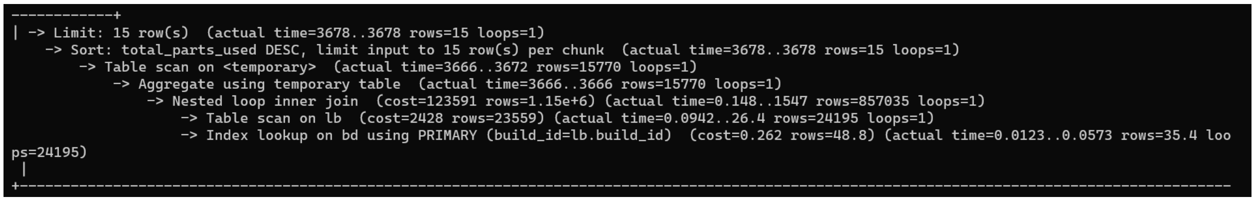


```
-----------+
| -> Limit: 15 row(s)  (actual time=3680..3680 rows=15 loops=1)
    -> Sort: total_parts_used DESC, limit input to 15 row(s) per chunk  (actual time=3680..3680 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=3668..3675 rows=15770 loops=1)
            -> Aggregate using temporary table  (actual time=3668..3668 rows=15770 loops=1)
                -> Nested loop inner join  (cost=123591 rows=1.15e+6) (actual time=0.161..1553 rows=857035 loops=1)
                    -> Table scan on lb  (cost=2428 rows=23559) (actual time=0.104..24.8 rows=24195 loops=1)
                    -> Index lookup on bd using PRIMARY (build_id=lb.build_id)  (cost=0.262 rows=48.8) (actual time=0.0125..0.0577 rows=35.4 loo
ps=24195)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------
```

Index 1:
Index on Build_Details.part_quantity

Why: This index will enhance the performance of queries that join the Build_Details table with the BUILDS table, especially when retrieving part quantities.

CREATE INDEX idx_part_details ON Build_Details (part_quantity);



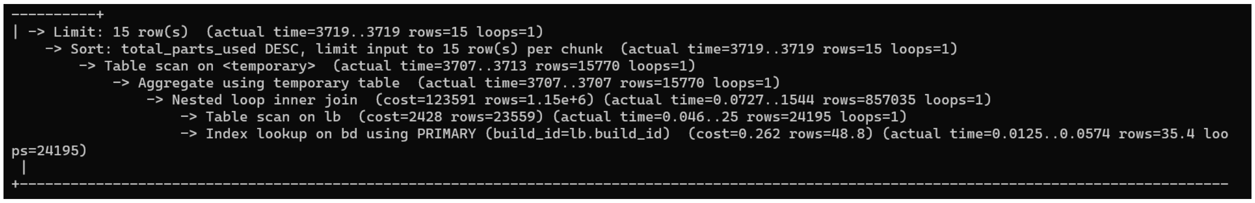
```
------------+
| -> Limit: 15 row(s)  (actual time=3678..3678 rows=15 loops=1)
    -> Sort: total_parts_used DESC, limit input to 15 row(s) per chunk  (actual time=3678..3678 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=3666..3672 rows=15770 loops=1)
            -> Aggregate using temporary table  (actual time=3666..3666 rows=15770 loops=1)
                -> Nested loop inner join  (cost=123591 rows=1.15e+6) (actual time=0.148..1547 rows=857035 loops=1)
                    -> Table scan on lb  (cost=2428 rows=23559) (actual time=0.0942..26.4 rows=24195 loops=1)
                    -> Index lookup on bd using PRIMARY (build_id=lb.build_id)  (cost=0.262 rows=48.8) (actual time=0.0123..0.0573 rows=35.4 loo
ps=24195)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------
```

Findings and Explanation: The actual execution time remained roughly the same. The query cost remained roughly the same. The indexing strategy did not optimize and the query decreased in speed with less efficient resource use.

Index 2:
Index on Builds.release_year

Why: This index will enhance the performance of queries that join the Build_Details table with the BUILDS table, especially when counting unique release across builds

CREATE INDEX idx_build_year ON BUILDS (build_release_year);



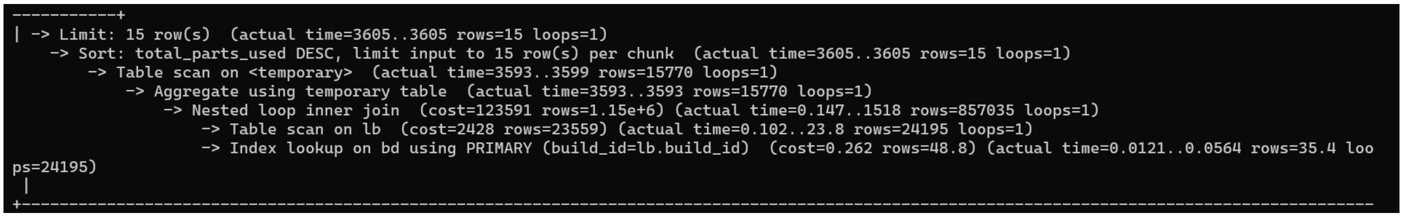
```
----------+
| -> Limit: 15 row(s)  (actual time=3719..3719 rows=15 loops=1)
    -> Sort: total_parts_used DESC, limit input to 15 row(s) per chunk  (actual time=3719..3719 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=3707..3713 rows=15770 loops=1)
            -> Aggregate using temporary table  (actual time=3707..3707 rows=15770 loops=1)
                -> Nested loop inner join  (cost=123591 rows=1.15e+6) (actual time=0.0727..1544 rows=857035 loops=1)
                    -> Table scan on lb  (cost=2428 rows=23559) (actual time=0.046..25 rows=24195 loops=1)
                    -> Index lookup on bd using PRIMARY (build_id=lb.build_id)  (cost=0.262 rows=48.8) (actual time=0.0125..0.0574 rows=35.4 loo
ps=24195)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------
```

Findings and Explanation: The actual execution time remained roughly the same. The query cost remained roughly the same. The indexing strategy did not optimize and the query decreased in speed with less efficient resource use.

Index 3:
Index on BUILDS.age_rating

Why: This composite index will enhance the performance of queries that join the Build_Details table with the BUILDS table, especially when retrieving part quantities and counting unique parts across builds

CREATE INDEX idx_build_age ON BUILDS(build_age_rating);



```
-----------+
| -> Limit: 15 row(s)  (actual time=3605..3605 rows=15 loops=1)
    -> Sort: total_parts_used DESC, limit input to 15 row(s) per chunk  (actual time=3605..3605 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=3593..3599 rows=15770 loops=1)
            -> Aggregate using temporary table  (actual time=3593..3593 rows=15770 loops=1)
                -> Nested loop inner join  (cost=123591 rows=1.15e+6) (actual time=0.147..1518 rows=857035 loops=1)
                    -> Table scan on lb  (cost=2428 rows=23559) (actual time=0.102..23.8 rows=24195 loops=1)
                    -> Index lookup on bd using PRIMARY (build_id=lb.build_id)  (cost=0.262 rows=48.8) (actual time=0.0121..0.0564 rows=35.4 loo
ps=24195)
 |
+------------------------------------------------------------------------------------------------------------------------------------------
```

Findings and Explanation: The actual execution time is roughly the same. The query cost remained roughly the same. The indexing strategy did not optimize and the query decreased in speed with less efficient resource use.

Final Choice

After evaluating the various indexing configurations, we ultimately selected the index on part_quantity for BUILD_DETAILS table and combined index on build_year_release and build_rating for BUILDS table and index on part_name for PARTS table as our final design. This decision was based on the consistent performance gains observed across multiple queries, particularly in those that build_release_year and build_rating the most. The improvements were significant enough to justify the added overhead of maintaining the index, and we anticipate that it will enhance user experience by reducing query response times.