DDL Commands

```
Create Table USERS (
        user_id VARCHAR(100) Primary key,
        username VARCHAR(100),
        password VARCHAR(100),
        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)
        );
        LOAD DATA LOCAL INFILE 'storage-legolab/data/PARTS.csv'' INTO TABLE
table_name FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES
TERMINATED BY '\r\n'; IGNORE 1 ROWS;

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 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 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 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)
```

```
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)
```

Count of Each Table

```
mysql> SELECT COUNT(*) FROM PARTS;
+----------+
| COUNT(*) |
+----------+
|    77949 |
+----------+
1 row in set (10.25 sec)
```

```
mysql> SELECT COUNT(*) FROM BUILDS;
+----------+
| COUNT(*) |
+----------+
|    24196 |
+----------+
1 row in set (4.79 sec)
```

```
mysql> SELECT COUNT(*) FROM THEMES;
+----------+
| COUNT(*) |
+----------+
|     1046 |
+----------+
1 row in set (0.01 sec)
```

```
mysql> SELECT COUNT(*) FROM USERS;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.08 sec)
```

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

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

**Note - We had to use small versions of these tables with 5000 to 10000 rows in order to actually get outputs within a decent amount of time and not have to run every single query for hours.

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 small_PARTS AS lp
JOIN small_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 small_PARTS AS lp
    -> JOIN small_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 |
+----------+-----------------------------------------------------------+---------------------+
| 2357     | Brick 2 x 2 Corner                                        |               30118 |
| 2339     | Brick Arch 1 x 5 x 4 [Continuous Bow]                     |                 870 |
| 2343     | Equipment Goblet / Glass                                  |                 870 |
| 2335     | Flag 2 x 2 Square [Thin Clips]                            |                 728 |
| 2356     | Brick 4 x 6                                               |                 290 |
| 2340     | Tail 4 x 1 x 3                                            |                 128 |
| 132a     | Tyre Smooth Old Style - Small                             |                 126 |
| 2345     | Panel 3 x 3 x 6 Corner Wall                               |                  80 |
| 12825    | Tile Special 1 x 1 with Clip with Rounded Tips            |                  80 |
| 2039     | Lamp Post 2 x 2 x 7 with 6 Base Flutes                    |                  63 |
| 22667    | Plant, Cherries                                           |                  40 |
| 2341     | Slope Inverted 45Â° 3 x 1 Double with 2 Completely Open Studs |                  16 |
| 14226c11 | String with End Studs 11L Overall                         |                  16 |
| 23306    | Weapon Lightsaber Hilt without Bottom Ring                |                  12 |
| 2342     | Equipment Control Panel                                   |                   8 |
+----------+-----------------------------------------------------------+---------------------+
15 rows in set (0.03 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 small_BUILDS AS lb
JOIN small_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 small_BUILDS AS lb
    -> JOIN small_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 |
+----------+-----------------------------------+-------------------+
| 10184-1  | Town Plan                         |               402 |
| 10185-1  | Green Grocer                      |               297 |
| 10159-1  | City Airport                      |               293 |
| 10159-2  | City Airport (Full Size Image Box) |               293 |
| 10129-1  | Rebel Snowspeeder                 |               280 |
| 10179-1  | Millennium Falcon                 |               270 |
| 10019-1  | Rebel Blockade Runner             |               247 |
| 10134-1  | Y-wing Attack Starfighter         |               247 |
| 10144-1  | Sandcrawler                       |               220 |
| 10132-1  | Motorized Hogwarts Express        |               208 |
| 10030-1  | Imperial Star Destroyer           |               205 |
| 10123-1  | Cloud City                        |               184 |
| 10173-1  | Holiday Train                     |               183 |
| 10040-1  | Black Seas Barracuda              |               179 |
| 10176-1  | Royal King's Castle               |               178 |
+----------+-----------------------------------+-------------------+
15 rows in set (0.23 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 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;
```

```
mysql> SELECT build_id, build_name, build_release_year, build_rating FROM small_BUILDS WHERE build_release_year >= 2019 AND build_rating > (     SELECT AVG(bui
ld_rating)     FROM small_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 |
+-------------+-----------------------------------------------------+--------------------+--------------+
| 202292404-1 | Ninjago Arin Backpack with Gym Bag and Pencil Case | 2024               |         5.00 |
| 11038-1     | Vibrant Creative Brick Box                          | 2024               |         5.00 |
| 31147-1     | Retro Camera                                        | 2024               |         5.00 |
| 10794-1     | Team Spidey Web Spinner Headquarters                | 2024               |         5.00 |
| 202832409-1 | Ninjago Red Duffle Bag                              | 2024               |         4.90 |
| 30672-1     | Steve and Baby Panda                                | 2024               |         4.90 |
| 202792410-1 | Friends Nova and Aliya Backpack                     | 2024               |         4.80 |
| 21259-1     | The Pirate Ship Voyage                              | 2024               |         4.80 |
| 10795-1     | Crafting with Baby Box                              | 2024               |         4.80 |
| 242403-1    | Thor                                                | 2024               |         4.70 |
| 11036-1     | Creative Vehicles                                   | 2024               |         4.60 |
| 30665-1     | Baby Gorilla Encounter                              | 2024               |         4.60 |
| 2000409-2   | Window Exploration Bag                              | 2024               |         4.60 |
| 10332-1     | Medieval Town Square                                | 2024               |         4.50 |
| 122403-1    | Owen with Helicopter                                | 2024               |         4.50 |
+-------------+-----------------------------------------------------+--------------------+--------------+
15 rows in set (0.01 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 small_BUILDS AS lb
JOIN small_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;
```

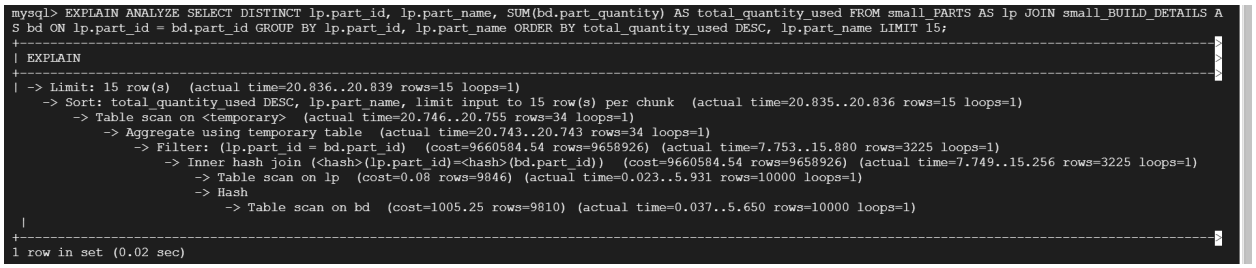
```
mysql> SELECT lb.build_id, lb.build_name, SUM(bd.part_quantity) AS total_parts_used
    -> FROM small_BUILDS AS lb
    -> JOIN small_BUILD_DETAILS AS bd ON lb.build_id = bd.build_id
    -> GROUP BY lb.build_id, lb.build_name
    -> ORDER BY total_parts_used DESC
    -> LIMIT 15;
+----------+---------------------------+------------------+
| build_id | build_name                | total_parts_used |
+----------+---------------------------+------------------+
| 10179-1  | Millennium Falcon         |             4983 |
| 10143-1  | Death Star II             |             3461 |
| 10181-1  | Eiffel Tower 1:300        |             2875 |
| 10030-1  | Imperial Star Destroyer   |             2804 |
| 10185-1  | Green Grocer              |             2302 |
| 10184-1  | Town Plan                 |             1910 |
| 10018-1  | Darth Maul                |             1867 |
| 10019-1  | Rebel Blockade Runner     |             1741 |
| 10129-1  | Rebel Snowspeeder         |             1414 |
| 10134-1  | Y-wing Attack Starfighter |             1352 |
| 10144-1  | Sandcrawler               |             1322 |
| 10175-1  | Vader's TIE Advanced      |             1212 |
| 10177-1  | Boeing 787 Dreamliner     |             1102 |
| 10183-1  | Hobby Train               |             1070 |
| 10178-1  | Motorized Walking AT-AT   |              991 |
+----------+---------------------------+------------------+
15 rows in set (0.03 sec)
```

INDEXING:

Query 1

Default Index

- Cost: 1005.25 for table scan
- Time: 5.65 on table scan

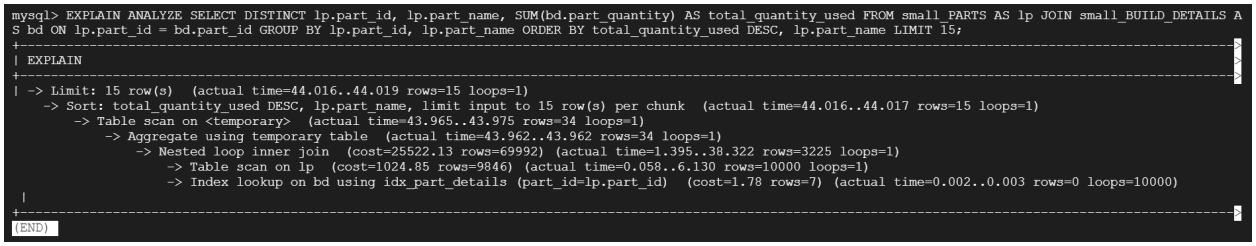


Index 1:

Index on Build_Details.part_id

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 small_BUILD_DETAILS (part_id);

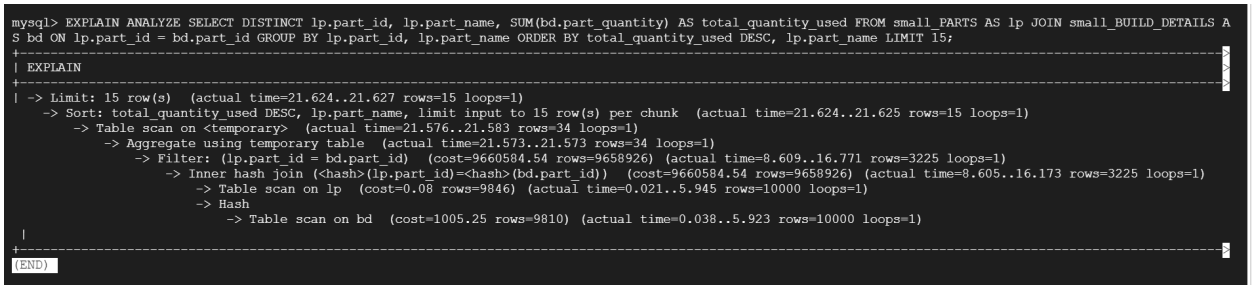


Findings and Explanation: The actual execution time decreased from 5.65 seconds to 0.002 seconds, showing a measurable improvement. The query cost decreased considerably to 1.78 from 1005. The indexing strategy effectively optimized the query by improving the speed of joins, aggregations, and sorting operations, leading to faster execution times and more efficient resource usage.

Index 2:
Index on Build_Details.build_id

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

CREATE INDEX idx_build_details ON Build_Details (build_id);



Findings and Explanation: The actual execution time increased from 5.65 seconds to 5.78 seconds, showing a measurable improvement. The query cost remained roughly the same. The indexing strategy is not that great compared to the default indexing
Index 3:
Index on Build_Details.part_id and Build_Details.build_id

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_part_build_details ON small_BUILD_DETAILS(part_id, build_id);

```
mysql> EXPLAIN ANALYZE SELECT DISTINCT lp.part_id, lp.part_name, SUM(bd.part_quantity) AS total_quantity_used FROM small_PARTS AS lp JOIN small_BUILD_DETAILS A
S 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;
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                |
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=41.054..41.057 rows=15 loops=1)
    -> Sort: total_quantity_used DESC, lp.part_name, limit input to 15 row(s) per chunk  (actual time=41.053..41.055 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=40.988..41.001 rows=34 loops=1)
            -> Aggregate using temporary table  (actual time=40.983..40.983 rows=34 loops=1)
                -> Nested loop inner join  (cost=25522.13 rows=69992) (actual time=0.443..35.535 rows=3225 loops=1)
                    -> Table scan on lp  (cost=1024.85 rows=9846) (actual time=0.043..6.253 rows=10000 loops=1)
                    -> Index lookup on bd using idx_part_build_details (part_id=lp.part_id)  (cost=1.78 rows=7) (actual time=0.002..0.003 rows=0 loops=10000)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
(END)
```

Findings and Explanation: The actual execution time decreased from 5.65 seconds to 0.03 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 2

Default Index

- Cost: 504.95 on scan
- Time: 44.07 on limit

```
mysql> EXPLAIN ANALYZE SELECT lb.build_id, lb.build_name, COUNT(DISTINCT bd.part_id, bd.part_color) AS unique_part_count FROM small_BUILDS AS lb JOIN small_BUI
LD_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;
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                |
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=44.067..44.070 rows=15 loops=1)
    -> Sort: unique_part_count DESC, lb.build_name, limit input to 15 row(s) per chunk  (actual time=44.066..44.069 rows=15 loops=1)
        -> Stream results  (actual time=30.724..43.914 rows=158 loops=1)
            -> Group aggregate: count(distinct small_BUILD_DETAILS.part_id,small_BUILD_DETAILS.part_color)  (actual time=30.720..43.853 rows=158 loops=1)
                -> Sort: lb.build_id, lb.build_name  (actual time=30.703..31.829 rows=10000 loops=1)
                    -> Stream results  (cost=4716554.17 rows=4715667) (actual time=4.858..24.330 rows=10000 loops=1)
                        -> Filter: (bd.build_id = lb.build_id)  (cost=4716554.17 rows=4715667) (actual time=4.851..19.438 rows=10000 loops=1)
                            -> Inner hash join (<hash>(bd.build_id)=<hash>(lb.build_id))  (cost=4716554.17 rows=4715667) (actual time=4.848..17.014 rows=10000
                                -> Table scan on bd  (cost=0.10 rows=9810) (actual time=0.019..6.137 rows=10000 loops=1)
                                -> Hash
                                    -> Table scan on lb  (cost=504.95 rows=4807) (actual time=0.032..3.205 rows=5000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
(END)
```

Index 1:

Index on Build_Details.part_id

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_id);

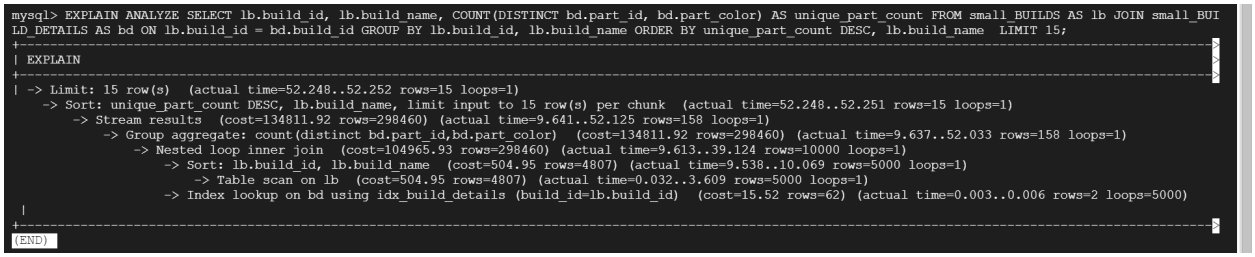
```
mysql> EXPLAIN ANALYZE SELECT lb.build_id, lb.build_name, COUNT(DISTINCT bd.part_id, bd.part_color) AS unique_part_count FROM small_BUILDS AS lb JOIN small_BUI
LD_DETAILS AS bd ON lb.build_id = bd.build_id GROUP BY lb.build_id, lb.build_name ORDER BY unique_part_count DESC, lb.build_name  LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                |
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=44.602..44.606 rows=15 loops=1)
    -> Sort: unique_part_count DESC, lb.build_name, limit input to 15 row(s) per chunk  (actual time=44.601..44.604 rows=15 loops=1)
        -> Stream results  (actual time=30.957..44.501 rows=158 loops=1)
            -> Group aggregate: count(distinct small_BUILD_DETAILS.part_id,small_BUILD_DETAILS.part_color)  (actual time=30.953..44.385 rows=158 loops=1)
                -> Sort: lb.build_id, lb.build_name  (actual time=30.934..32.060 rows=10000 loops=1)
                    -> Stream results  (cost=4716554.17 rows=4715667) (actual time=4.604..24.320 rows=10000 loops=1)
                        -> Filter: (bd.build_id = lb.build_id)  (cost=4716554.17 rows=4715667) (actual time=4.596..19.425 rows=10000 loops=1)
                            -> Inner hash join (<hash>(bd.build_id)=<hash>(lb.build_id))  (cost=4716554.17 rows=4715667) (actual time=4.592..17.077 rows=10000
                                -> Table scan on bd  (cost=0.10 rows=9810) (actual time=0.019..6.351 rows=10000 loops=1)
                                -> Hash
                                    -> Table scan on lb  (cost=504.95 rows=4807) (actual time=0.029..2.970 rows=5000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------+
(END)
```

Findings and Explanation: The actual execution time decreased from 44.07 seconds to 2.97 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 Build_Details.build_id

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

CREATE INDEX idx_build_details ON Build_Details (build_id);



Findings and Explanation: The actual execution time decreased from 44.07 seconds to 3.609 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 Build_Details.part_id and Build_Details.build_id

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_part_build_details ON Build_Details (part_id, build_id);

```
mysql> EXPLAIN ANALYZE SELECT lb.build_id, lb.build_name, COUNT(DISTINCT bd.part_id, bd.part_color) AS unique_part_count FROM small_BUILDS AS lb JOIN small_BUI
LD_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;
+----------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                  |
+----------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=43.947..43.950 rows=15 loops=1)
    -> Sort: unique_part_count DESC, lb.build_name, limit input to 15 row(s) per chunk  (actual time=43.947..43.949 rows=15 loops=1)
        -> Stream results  (actual time=30.586..43.869 rows=158 loops=1)
            -> Group aggregate: count(distinct small_BUILD_DETAILS.part_id,small_BUILD_DETAILS.part_color)  (actual time=30.582..43.804 rows=158 loops=1)
                -> Sort: lb.build_id, lb.build_name  (actual time=30.561..31.671 rows=10000 loops=1)
                    -> Stream results  (cost=4716554.17 rows=4715667) (actual time=4.472..24.333 rows=10000 loops=1)
                        -> Filter: (bd.build_id = lb.build_id)  (cost=4716554.17 rows=4715667) (actual time=4.466..19.266 rows=10000 loops=1)
                            -> Inner hash join (<hash>(bd.build_id)=<hash>(lb.build_id))  (cost=4716554.17 rows=4715667) (actual time=4.463..16.914 rows=10000
                                -> Table scan on bd  (cost=0.10 rows=9810) (actual time=0.017..6.265 rows=10000 loops=1)
                                -> Hash
                                    -> Table scan on lb  (cost=504.95 rows=4807) (actual time=0.034..2.994 rows=5000 loops=1)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------------+
(END)
```

Findings and Explanation: The actual execution time decreased from 44.07 seconds to 2.994 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);

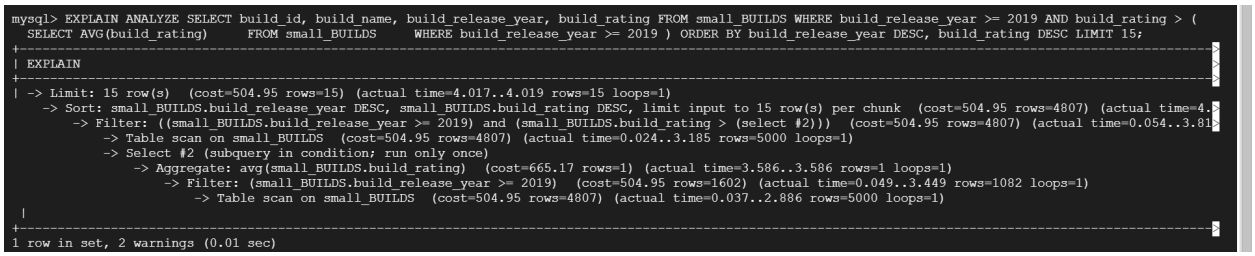
```
mysql> EXPLAIN ANALYZE SELECT build_id, build_name, build_release_year, build_rating FROM small_BUILDS WHERE build_release_year >= 2019 AND build_rating > (
  SELECT AVG(build_rating)     FROM small_BUILDS     WHERE build_release_year >= 2019 ) ORDER BY build_release_year DESC, build_rating DESC 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 small_BUILDS(build_rating);

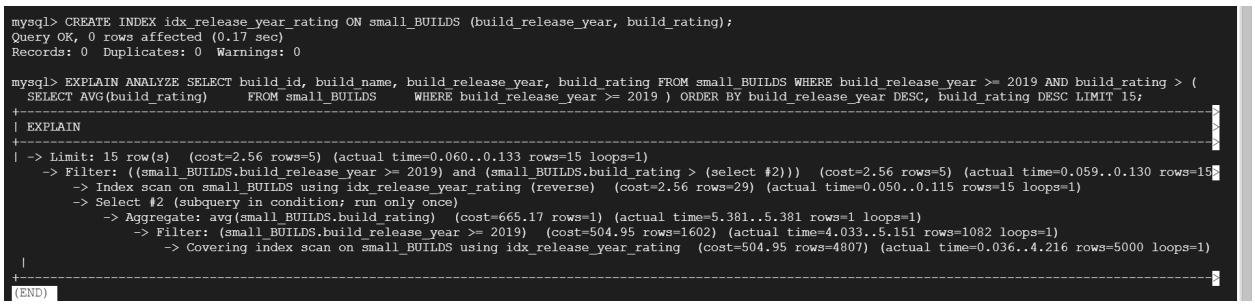


Findings and Explanation: The actual execution time remained the same. The query cost actually increased. The indexing strategy reduced the speed of the query by decreasing the speed of joins, aggregations, and sorting operations, leading to slower execution times and less efficient resource usage.

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);



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: 504.95 on table scan
- Time: 29.573 on limit

```
mysql> EXPLAIN ANALYZE SELECT lb.build_id, lb.build_name, SUM(bd.part_quantity) AS total_parts_used FROM small_BUILDS AS lb JOIN small_BUILD_DETAILS AS bd ON l
b.build_id = bd.build_id GROUP BY lb.build_id, lb.build_name ORDER BY total_parts_used DESC LIMIT 15;
+-----------------------------------------------------------------------------------------------------------------------------------------------
| EXPLAIN
+-----------------------------------------------------------------------------------------------------------------------------------------------
| -> Limit: 15 row(s)  (actual time=29.571..29.573 rows=15 loops=1)
    -> Sort: total_parts_used DESC, limit input to 15 row(s) per chunk  (actual time=29.570..29.571 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=29.450..29.510 rows=158 loops=1)
            -> Aggregate using temporary table  (actual time=29.446..29.446 rows=158 loops=1)
                -> Filter: (bd.build_id = lb.build_id)  (cost=4716554.17 rows=4715667) (actual time=4.748..18.837 rows=10000 loops=1)
                    -> Inner hash join (<hash>(bd.build_id)=<hash>(lb.build_id))  (cost=4716554.17 rows=4715667) (actual time=4.745..16.389 rows=10000 loops=1)
                        -> Table scan on bd  (cost=0.10 rows=9810) (actual time=0.020..5.925 rows=10000 loops=1)
                        -> Hash
                            -> Table scan on lb  (cost=504.95 rows=4807) (actual time=0.031..3.172 rows=5000 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------
(END)
```

Index 1:
Index on Build_Details.part_id

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

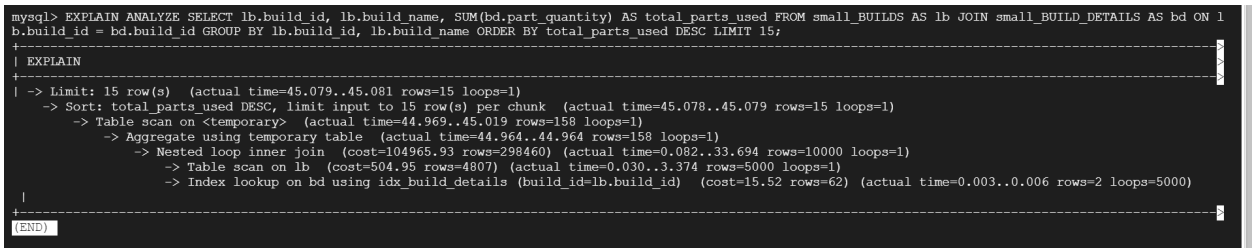
CREATE INDEX idx_part_details ON Build_Details (part_id);

```
mysql> EXPLAIN ANALYZE SELECT lb.build_id, lb.build_name, SUM(bd.part_quantity) AS total_parts_used FROM small_BUILDS AS lb JOIN small_BUILD_DETAILS AS bd ON l
b.build_id = bd.build_id GROUP BY lb.build_id, lb.build_name ORDER BY total_parts_used DESC LIMIT 15;
+-----------------------------------------------------------------------------------------------------------------------------------------------
| EXPLAIN
+-----------------------------------------------------------------------------------------------------------------------------------------------
| -> Limit: 15 row(s)  (actual time=36.705..36.709 rows=15 loops=1)
    -> Sort: total_parts_used DESC, limit input to 15 row(s) per chunk  (actual time=36.704..36.706 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=36.507..36.594 rows=158 loops=1)
            -> Aggregate using temporary table  (actual time=36.479..36.479 rows=158 loops=1)
                -> Filter: (bd.build_id = lb.build_id)  (cost=4716554.17 rows=4715667) (actual time=4.610..23.706 rows=10000 loops=1)
                    -> Inner hash join (<hash>(bd.build_id)=<hash>(lb.build_id))  (cost=4716554.17 rows=4715667) (actual time=4.606..20.074 rows=10000 loops=1)
                        -> Table scan on bd  (cost=0.10 rows=9810) (actual time=0.018..6.959 rows=10000 loops=1)
                        -> Hash
                            -> Table scan on lb  (cost=504.95 rows=4807) (actual time=0.028..3.071 rows=5000 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------
(END)
```

Findings and Explanation: The actual execution time decreased from 29.573 seconds to 36.709 seconds, showing increased runtime. The query cost remained roughly the same. The indexing strategy was not optimized and the query decreased in speed with less efficient resource use.
Index 2:
Index on Build_Details.build_id

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

CREATE INDEX idx_build_details ON Build_Details (build_id);

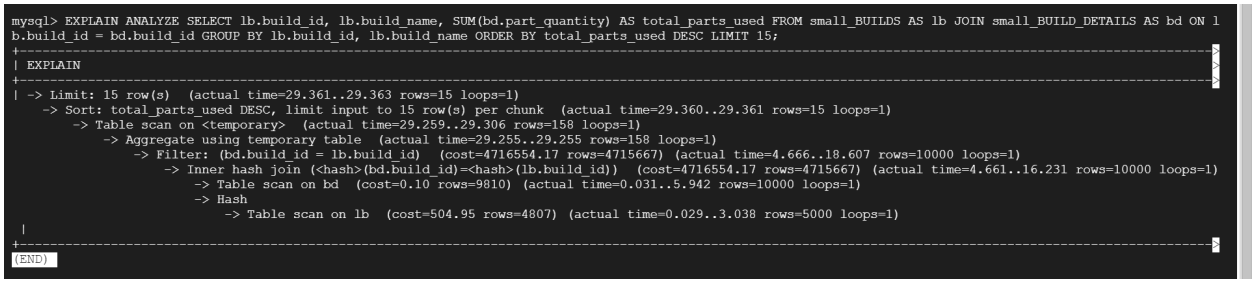


Findings and Explanation: The actual execution time decreased from 29.573 seconds to 45.079 seconds, showing increased runtime. The query cost remained roughly the same. The indexing strategy was not optimized and the query decreased in speed with less efficient resource use.

Index 3:
Index on Build_Details.part_id and Build_Details.build_id

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_part_build_details ON Build_Details (part_id, build_id);



Findings and Explanation: The actual execution time decreased from 29.573 seconds to 29.363 seconds, showing a slight 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.

Final Choice

After evaluating the various indexing configurations, we ultimately selected the index on the combined index on part_id and build_id for build_details for BUILD_DETAILS table and combined index on build_year_release and build_rating for BUILDS 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.