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

```
mysql> SELECT COUNT(*) FROM SUPPLIERS;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.04 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 lp.part_id, lp.part_color, 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_color, lp.part_name
ORDER BY total_quantity_used DESC, lp.part_name
LIMIT 15;
```

2. Query: List Themes with the Highest Average Build Rating

This query calculates the average rating for each theme by joining Build_Themes with BUILDS and grouping by theme. The output is ordered by average rating in descending order.

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

```
SELECT bt.theme_name, AVG(lb.build_rating) AS avg_rating FROM Build_Themes AS
bt JOIN BUILDS AS lb ON bt.theme_id = lb.build_id GROUP BY bt.theme_name
ORDER BY avg_rating DESC, bt.theme_name LIMIT 15;
```

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_ratin
g)     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^M             |         5.00 |
| 53434-1         | Spaceman Red Pencil Case Pop Up                                              | 2025^M             |         5.00 |
| 53500-1         | Space Bus Molded Pencil Case                                                 | 2025^M             |         4.90 |
| 77002-1         | Cyclone vs. Metal Sonic                                                      | 2025^M             |         4.90 |
| 9780241727416-1 | LEGO Ideas Activity Book: Animals                                            | 2025^M             |         4.60 |
| 77053-1         | Stargazing with Celeste                                                      | 2025^M             |         4.20 |
| 9780241740859-1 | Our Amazing Universe: Fantastic Building Ideas and Facts About Our Universe  | 2025^M             |         4.10 |
| 9781837250639-1 | ReBuild Activity Cards: Magic                                                | 2025^M             |         4.00 |
| 9780794453343-1 | Build and Play! Easter                                                       | 2025^M             |         3.80 |
| 72035-1         | Mario Kart – Toad's Garage                                                   | 2025^M             |         3.70 |
| 910043-1        | Forest Stronghold                                                            | 2025^M             |         3.50 |
| 53389-1         | Spaceman Blue Pencil Case Pop Up                                             | 2025^M             |         3.50 |
| 910042-1        | Lost City                                                                    | 2025^M             |         3.30 |
| 77001-1         | Sonic’s Campfire Clash                                                       | 2025^M             |         3.00 |
| 53325-1         | Ninjago Kai Molded Pencil Case                                               | 2025^M             |         2.60 |
+-----------------+------------------------------------------------------------------------------+--------------------+--------------+
15 rows in set, 2 warnings (0.81 sec)
```

4. 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;

5. 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;
```

INDEXING:

Query 1

Default Index

- Cost: 802.05
- Time: 7.836..7.840

Index 1:

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 ON Build_Details (part_id, build_id);
```

Findings and Explanation: The actual execution time decreased from 6.296 seconds to 4.126 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.build_release_year and BUILDS.build_rating

Why: An index on these columns will speed up queries that filter on build_release_year and those that check for build_rating, like the one that identifies builds released within a certain timeframe or above a certain rating.

```
CREATE INDEX idx_release_year_rating ON BUILDS (build_release_year,
build_rating);
```

Findings and Explanation: The actual execution time decreased from 7.428 seconds to 6.136 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_Themes.theme_id

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_theme_id ON Build_Themes (theme_id);

Findings and Explanation: The actual execution time decreased from 6.695 seconds to 5.187 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: 562.05
- Time: 7.836..7.840

Index 1:
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 ON Build_Details (part_id, build_id);

Findings and Explanation: The actual execution time decreased from 6.296 seconds to 4.126 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.build_release_year and BUILDS.build_rating

Why: An index on these columns will speed up queries that filter on build_release_year and those that check for build_rating, like the one that identifies builds released within a certain timeframe or above a certain rating.

CREATE INDEX idx_release_year_rating ON BUILDS (build_release_year, build_rating);

Findings and Explanation: The actual execution time decreased from 7.428 seconds to 6.136 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_Themes.theme_id

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_theme_id ON Build_Themes (theme_id);

Findings and Explanation: The actual execution time decreased from 5.288 seconds to 5.233 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: 1983.10

- Time: 74.770…74.775

```
mysql> EXPLAIN ANALYZE 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
    ->
    -> ;
+-------------------------------------------------------------------------------------------------------------------------------------------------------->
| EXPLAIN                                                                                                                                                  >
+-------------------------------------------------------------------------------------------------------------------------------------------------------->
| -> Limit: 15 row(s)  (cost=1983.10 rows=15) (actual time=74.770..74.775 rows=15 loops=1)
    -> Sort: BUILDS.build_release_year DESC, BUILDS.build_rating DESC, limit input to 15 row(s) per chunk  (cost=1983.10 rows=24362) (actual time=74.768..74.772>
        -> Filter: ((BUILDS.build_release_year >= 2019) and (BUILDS.build_rating > (select #2)))  (cost=1983.10 rows=24362) (actual time=28.237..56.315 rows=339>
            -> Table scan on BUILDS  (cost=1983.10 rows=24362) (actual time=0.831..23.613 rows=24196 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Aggregate: avg(BUILDS.build_rating)  (cost=3336.44 rows=1) (actual time=24.226..24.228 rows=1 loops=1)
                    -> Filter: (BUILDS.build_release_year >= 2019)  (cost=2524.45 rows=8120) (actual time=0.092..22.439 rows=6880 loops=1)
                        -> Table scan on BUILDS  (cost=2524.45 rows=24362) (actual time=0.083..17.359 rows=24196 loops=1)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------->
(END)
```

Index 1:
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 ON Build_Details (part_id, build_id);

Findings and Explanation: The actual execution time decreased from 5.288 seconds to 5.233 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.build_release_year and BUILDS.build_rating

Why: An index on these columns will speed up queries that filter on build_release_year and those that check for build_rating, like the one that identifies builds released within a certain timeframe or above a certain rating.

CREATE INDEX idx_release_year_rating ON BUILDS (build_release_year, build_rating);

Findings and Explanation: The actual execution time decreased from 6.296 seconds to 4.126 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_Themes.theme_id

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_theme_id ON Build_Themes (theme_id);

Findings and Explanation: The actual execution time decreased from 5.288 seconds to 5.233 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 4
Default Index

- Cost: 802.05
- Time: 7.836..7.840

Index 1:
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 ON Build_Details (part_id, build_id);

Findings and Explanation: The actual execution time decreased from 5.288 seconds to 5.233 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.build_release_year and BUILDS.build_rating

Why: An index on these columns will speed up queries that filter on build_release_year and those that check for build_rating, like the one that identifies builds released within a certain timeframe or above a certain rating.

CREATE INDEX idx_release_year_rating ON BUILDS (build_release_year, build_rating);

Findings and Explanation: The actual execution time decreased from 5.288 seconds to 5.233 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_Themes.theme_id

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_theme_id ON Build_Themes (theme_id);

Findings and Explanation: The actual execution time decreased from 6.296 seconds to 4.126 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.

Final Choice

After evaluating the various indexing configurations, we ultimately selected the index on the combined index on part_id and build_id as our final design. This decision was based on the consistent performance gains observed across multiple queries, particularly in those that part id and build id 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.