Feature	Single-row Subquery	Correlated Subquery
Execution	Runs **once**	Runs **once per outer row**
Dependence	Independent of outer query	References outer query columns
Example Use	Compare salary > company avg	Compare salary > dept avg
Performance	Faster (1 execution)	Slower (many executions possible)

👉 Quick **interview tip**:

- If you see the inner subquery referring to an **outer alias (like f.department_id)** → it's **correlated**.
- If not → it's a **single-row (or multi-row) subquery**.

Feature	Multi-Row Subquery	Multi-Column Subquery
Definition	Subquery that returns **one column with multiple rows**	Subquery that returns **multiple columns (tuples)** with multiple rows
Operators Used	`IN`, `ANY`, `ALL`	Tuple comparison with `IN` or `EXISTS`
Outer Query Comparison	Compares **a single column** from outer query against a set of values from inner query	Compares **multiple columns together** as a single tuple
Example	`sql SELECT first_name, fare FROM titanic WHERE fare > ANY ( SELECT fare FROM titanic WHERE pclass = 1 );` → Compares one column (`fare`) to many fares.	`sql SELECT * FROM titanic e WHERE (e.pclass, e.embark_town) IN ( SELECT t.pclass, t.embark_town FROM titanic t WHERE survived = 1 );` → Compares two columns `(pclass, embark_town)` as a pair.

Quick Identifier	Inner query: `SELECT col1 ...`	Inner query: `SELECT col1, col2 ...` Outer query: `(col1, col2)` in tuple form
Use Case	- Find values in a single column that match multiple rows (e.g., fares, ages, IDs).	- Match on **combinations** of values (e.g., class + embark_town, dept_id + job_id).

Quick Differentiation Trick

- **Look at the subquery SELECT**:
  - If it has **1 column** → Multi-Row.
  - If it has **2+ columns** → Multi-Column.
- Outer query will use either:
  - `col IN (SELECT col …)` → Multi-Row.
  - `(col1, col2) IN (SELECT col1, col2 …)` → Multi-Column.

Feature	**NOT IN**	**NOT EXISTS**
How it works	Compares a value to a **list** returned by the subquery.	Checks whether the subquery returns **any row at all**.
Syntax	`sql WHERE col NOT IN (SELECT col FROM table …)`	`sql WHERE NOT EXISTS (SELECT 1 FROM table WHERE …)`
NULL behavior	⚠️ If the subquery returns **even one NULL**, the whole `NOT IN` returns **no rows**	✅ Ignores NULLs automatically. Works safely even if subquery has NULLs.

	(because comparisons with NULL = unknown).	
Performance	Usually fine for small sets; can be slower if subquery list is big.	Often faster in large tables, since it stops at the **first match**.
Use case	Use when you're sure the subquery column **doesn't contain NULLs** (e.g., primary keys).	Use when NULLs are possible, or when you want to be 100% safe.
Example	Passengers not on decks where survivors were: `sql SELECT * FROM titanic e WHERE e.deck NOT IN ( SELECT deck FROM titanic WHERE survived=1 AND deck IS NOT NULL );`	Same query safe with `NOT EXISTS`: `sql SELECT * FROM titanic e WHERE NOT EXISTS ( SELECT 1 FROM titanic t WHERE t.deck = e.deck AND t.survived=1 );`

Quick rules to remember:

- ✅ **Prefer `NOT EXISTS`** → safer (handles NULLs correctly).
- ✅ **`NOT IN`** → only when you're 100% sure the subquery column has **no NULL values** (like IDs, primary keys).
- Many interviewers like to test this: *"What happens if the subquery returns NULL?"* — the answer: `NOT IN` fails, `NOT EXISTS` works fine.

Feature	Correlated Subquery	Multi-Row Subquery
Definition	Inner subquery depends on a column from the outer query. Runs **once for each row** of the outer query.	Inner subquery returns **multiple rows** (a set) that the outer query compares against. Runs **once total**, independently.

Execution	Repeated execution per outer row.	Single execution, returns a list/set.
Operators used	Normal comparison operators (`=`, `>`, `<`) with correlation to outer alias.	`IN`, `ANY`, `ALL` are commonly used to handle multiple values.
Example (Titanic dataset)	Passengers who paid more than the **average fare of their embark_town**: `sql SELECT f.passenger_no, f.fare FROM titanic f WHERE f.fare > ( SELECT AVG(t.fare) FROM titanic t WHERE t.embark_town = f.embark_town );` → inner query uses `f.embark_town` from the outer query → **correlated**.	Passengers in the **same pclass as survivors**: `sql SELECT passenger_no, first_name, pclass, survived FROM titanic WHERE pclass IN ( SELECT DISTINCT pclass FROM titanic WHERE survived = 1 );` → inner query returns a **list of classes** → **multi-row**.
When to use	When you need a calculation or condition that changes **per row** of the outer query (e.g., compare against that row's department avg).	When you need to check if a value is **in a set of results** (e.g., department is in the list of departments that meet a condition).
Performance	Can be slower on big data (runs many times). Often rewritten with JOINs or window functions.	Usually faster, since inner query runs once and returns a set.

Quick trick to differentiate

- If the **inner query references outer query columns** → it's **correlated**.
- If the **inner query stands alone and returns multiple rows** → it's **multi-row**.

Feature	IN	EXISTS
Definition	Compares a value from the outer query to a **list of values** returned by the subquery.	Tests whether the subquery returns **at least one row**. Doesn't care about values.

Execution	Subquery runs first → returns a set → outer query checks if value is in that set.	Outer query runs, then for each row, subquery checks if a matching row exists (correlated).
Return Type	Subquery must return **one column** (can be many rows).	Subquery can return **any column(s)**, SQL only checks row existence.
NULL behavior	⚠️ If subquery contains even **one NULL**, `NOT IN` may return no rows at all (because `col NOT IN (… , NULL)` is always unknown).	✅ `NOT EXISTS` safely handles NULLs (it only checks existence of rows, not values).
Performance	Better for small static lists (e.g., `WHERE col IN (1,2,3)`).	Often better for large, correlated queries (stops after finding the first match).
Example (Titanic dataset)	Passengers in the same pclass as survivors: `sql SELECT passenger_no, pclass FROM titanic WHERE pclass IN ( SELECT DISTINCT pclass FROM titanic WHERE survived = 1 );`	Passengers in the same pclass as survivors: `sql SELECT e.passenger_no, e.pclass FROM titanic e WHERE EXISTS ( SELECT 1 FROM titanic t WHERE t.pclass = e.pclass AND t.survived = 1 );`

Quick rules for interviews

- ✅ Use **IN** when comparing to a **list/set of values**.
- ✅ Use **EXISTS** when you just want to check if a matching row exists.
- ✅ Prefer **EXISTS** over `NOT IN` if NULLs might appear.

—--

Updating `rooms` vs `bookings` – why derived table was needed

Updating `rooms` – no problem

```
 UPDATE rooms r

SET r.status = 'High Demand'
```

```
WHERE EXISTS (
    SELECT b.roomnumber, COUNT(*) AS c
    FROM bookings b
    WHERE b.roomnumber = r.roomnumber
    GROUP BY b.roomnumber
    HAVING c > 5
);
```

1.
   - Here we **update `rooms`** but **read from `bookings`** in the subquery.
   - Different tables → **MySQL is happy**, no error.

Updating `bookings` – direct subquery causes error 1093

```
 UPDATE bookings
SET bookingstatus = 'High demand'
WHERE totalamount > (
    SELECT AVG(totalamount) FROM bookings   -- ❌ same table
);
```

2.
   - MySQL rule: **you can't update a table and read from the same table in a subquery** → error 1093.

Fix: use a *derived table* (temp result)

```
 UPDATE bookings c
SET c.bookingstatus = 'High demand'
WHERE EXISTS (
    SELECT roomnumber
    FROM (
        SELECT roomnumber, COUNT(*) AS c
        FROM bookings b
        GROUP BY roomnumber
        HAVING c > 5
    ) AS Raj              -- ✅ derived table
    WHERE Raj.roomnumber = c.roomnumber   -- ✅ correlation
);
```

3.
   - Inner query on `bookings` is materialized as **Raj** (a temp table).
   - Outer UPDATE reads from **Raj**, not directly from `bookings` → **no 1093 error**.
   - Correlation `Raj.roomnumber = c.roomnumber` ensures **only bookings of high-demand rooms** get updated.

🧠 **Memory line:**

When `UPDATE` and subquery use the **same table**, MySQL needs a **derived table (or JOIN)** to avoid error 1093.
Different table (like `rooms` reading `bookings`) → no problem.