Wednesday, 3 December 2025

Project Notes

Table Creation

- **UNSIGNED** prevents accidental inserts of negative values (EVERY TABLE)
- **DECIMAL(7,2)** limits price to 99999.99 (too small for some products) (PRODUCT, ORDER_PRODUCT)
- Mix **0** and **FALSE** (PRODUCT, USER)
- Most university systems run: (PRODUCT, ORDER_PRODUCT)
  - MySQL 5.7
  - OR MySQL 8.0.x in compatibility mode
    → CHECK constraints do **nothing**.
- **USER** is a reserved word in MySQL (USER)
- CHAR(13) UNIQUE **with dash**: YYYYMMDD-NNNN → 13 chars
- **Name**, **Password** are likely to collide with functions (PASSWORD() and NAME() exist in MySQL).
- Better avoid backticks like **'USER'** because:
  - you must ALWAYS escape them
  - queries become harder to read
  - errors are more likely
  - tools might break
- Specification: "Payment reference ... 64 character long string.", so PaymentReference VARCHAR(**64**) NOT NULL

Queries

Query 3

- Returns *extra* fields (**BasePriceNet, DiscountPercent**) and gives the price alias a different, less spec-aligned name (**FinalPriceWithVAT** instead of something like **CurrentRetailPrice**).

- DiscountPercent is **NOT NULL with default 0.**

That means **IFNULL(P.DiscountPercent, 0**) will always be the same as just P.DiscountPercent with the current design and data.

Query 4

- Forgot to select **Short Description** and **Current Retail Price.**

Query 5

- Needs **COALESCE**(ROUND(AVG(UP.Stars), 2), 0) AS AverageRating in order to return 0 average rating when a product has no reviews

Query 7

- The 30-day filter is correct, but it makes no visible difference because all our Orders have recent NOW() dates, so none are older than 30 days — the filter becomes meaningful only if some OrderDate values are more than 30 days old.

No order dates older than 30 days old

ProductID	Title	TotalUnitsSold
33	Smart RGB Bulb E27	5
10	BassTube Waterproof	4
3	EduBook Air 13	3
11	ErgoMaster 3000	2
7	OLED Cinema 65"	2
30	RCA to Aux Adapter	2
17	Active Stylus Gen 2	1
12	Architect Desk Lamp	1
21	Cast Iron Skillet	1
18	Folio Keyboard Case	1

Order dates older than 30 days old

ProductID	Title	TotalUnitsSold
3	EduBook Air 13	3
10	BassTube Waterproof	2
30	RCA to Aux Adapter	2
12	Architect Desk Lamp	1
21	Cast Iron Skillet	1
11	ErgoMaster 3000	1
15	GaN Fast Charger 100W	1
2	Hydra Gaming Tower	1
6	Nordic Commuter Backpack	1
7	OLED Cinema 65"	1

Take for example the **BassTubeWaterproof** product. Before the change on dates the company had sold 4 units **BassTubeWaterproof** of in the last 30 days but after the change the company sold 2 units **BassTubeWaterproof.**

Queries Analysis

Query 1:

'-> *Index lookup on Department using ParentDepartmentID (ParentDepartmentID = NULL), with index condition: (department.ParentDepartmentID is null) (cost=0.35 rows=1)*

'

- **EXPLAIN** shows that MySQL uses an **index lookup** on the Department table via the ParentDepartmentID index.
- The query efficiently retrieves rows using this existing index, so **no further optimization or additional indexing is needed**.

Query 2:

'-> *Sort: department.Title (cost=0.7 rows=2)*

-> Index lookup on Department using ParentDepartmentID (ParentDepartmentID = 1) (cost=0.7 rows=2)

'

- MySQL performs an **index lookup** using the existing `ParentDepartmentID` index to retrieve the matching departments efficiently.
- A small **sort operation** on `Title` occurs due to `ORDER BY`, but with only two rows this cost is negligible.
- The query is already optimized, and **no additional indexing is required**.

Query 3:

'-> *Sort: product.Title (cost=3.55 rows=33)*

-> Filter: (product.IsFeatured = true) (cost=3.55 rows=33)

-> Table scan on Product (cost=3.55 rows=33)

'

- MySQL scans **all 33 rows** of `Product` and then filters by `IsFeatured`.
- This is a **full table scan** (type = ALL), so the query is not optimized.

'-> *Index lookup on Product using idx_product_isfeatured_title (IsFeatured = true) (cost=1.75 rows=10)*
‘

- After adding the composite index idx_product_isfeatured_title (IsFeatured, Title), **EXPLAIN** shows an **index lookup** instead of a full table scan.

- MySQL now jumps directly to rows where `IsFeatured = TRUE` and reads them already ordered by `Title`.
- Rows examined drop from **33** to **10**, improving performance under the project's "rows examined" metric.

Query 4:

'*-> Sort: p.Title*

-> Table scan on <temporary> (cost=5.42..7.54 rows=5.89)

-> Temporary table with deduplication (cost=4.98..4.98 rows=5.89)

-> Nested loop inner join (cost=3.62 rows=5.89)

-> Nested loop inner join (cost=1.56 rows=5.89)

-> Covering index lookup on pk_base using PRIMARY (ProductID = (@ProductID)) (cost=0.45 rows=2)

-> Filter: (pk_other.ProductID <> <cache>((@ProductID))) (cost=0.398 rows=2.94)

-> Covering index lookup on pk_other using KeywordID (KeywordID = pk_base.KeywordID) (cost=0.398 rows=3.06)

-> Single-row index lookup on p using PRIMARY (ProductID = pk_other.ProductID) (cost=0.267 rows=1)

'

- MySQL performs **covering index lookups** on `Product_Keyword` for both the base product and related products, using existing B-tree indexes on `ProductID` and `KeywordID`.
- For each related product ID, the database performs a **single-row primary key lookup** on `Product`, which is the most efficient access method.
- A **temporary table with deduplication** is created due to the `DISTINCT` clause, followed by a small sort on `Title` required by `ORDER BY`.
- No full table scans occur on the base tables, and the execution plan is already efficient.

Query 5:

'*-> Sort: p.Title*

-> Stream results (cost=1.86 rows=1.41)

-> Group aggregate: avg(up.Stars) (cost=1.86 rows=1.41)

-> Nested loop left join (cost=1.4 rows=2)

-> Index lookup on p using DepartmentID (DepartmentID = (@DepartmentID)) (cost=0.7 rows=2)

-> Index lookup on up using PRIMARY (ProductID = p.ProductID) (cost=0.3 rows=1)

'

- **EXPLAIN** shows an **index lookup** on `Product.DepartmentID`, allowing MySQL to retrieve only products belonging to the selected department instead of scanning the whole table.
- For each product, MySQL performs a **single-row primary key lookup** on `User_Product` to compute the average rating.
- A small sort on `p.Title` is required due to `ORDER BY`, but the number of rows involved is minimal.
- No full table scans occur, and existing indexes already support the query efficiently, so **no additional indexing is needed**.

Query 6:

'-> Sort: product.DiscountPercent DESC, product.Title (cost=3.55 rows=33)

-> Filter: (product.DiscountPercent <> 0.00) (cost=3.55 rows=33)

-> Table scan on Product (cost=3.55 rows=33)

'

- **EXPLAIN** shows a **full table scan** on Product, examining all 33 rows before filtering products with DiscountPercent <> 0.
- MySQL then performs a separate **sort operation** on `(DiscountPercent DESC, Title)` because no index supports this ordering.
- This makes the query a strong candidate for optimization by adding a composite index.

'-> Sort: product.DiscountPercent DESC, product.Title (cost=2.37 rows=10)

-> Filter: (product.DiscountPercent <> 0.00) (cost=2.37 rows=10)

-> Covering index range scan on Product using idx_product_discount_title over (DiscountPercent < 0.00) OR (0.00 < DiscountPercent) (cost=2.37 rows=10)

‘

- After adding the composite index idx_product_discount_title (DiscountPercent, Title), **EXPLAIN** shows a **covering index range scan** instead of a full table scan.
- MySQL now examines only the rows that match the discount condition (**`DiscountPercent <> 0`**), reducing the estimated rows examined from **33 to 10**.
- Although a small sort step remains for `ORDER BY`, it now operates on a much smaller filtered set, improving query performance under the project’s “rows examined” metric.

Query 7:

'-> Limit: 10 row(s)

-> Sort: TotalUnitsSold DESC, p.Title, limit input to 10 row(s) per chunk

-> Table scan on <temporary>

-> Aggregate using temporary table

-> Nested loop inner join (cost=3.98 rows=4.67)

-> Nested loop inner join (cost=2.35 rows=4.67)

-> Filter: (o.OrderDate >= <cache>((curdate() - interval 30 day))) (cost=0.717 rows=4.67)

-> Table scan on o (cost=0.717 rows=14)

-> Index lookup on op using PRIMARY (OrderID = o.OrderID) (cost=0.271 rows=1)

-> Single-row index lookup on p using PRIMARY (ProductID = op.ProductID) (cost=0.271 rows=1)

‘

- **EXPLAIN** shows a **full table scan** on Orders before applying the date filter, so all rows are examined.
- The joins on Order_Product and Product use **index lookups**, so those parts are already efficient.
- Grouping, sorting, and LIMIT are expected; the main optimization opportunity is to **index OrderDate** to reduce examined rows.

Results with the original dataset (no older orders)

'*-> Limit: 10 row(s)*
-> Sort: TotalUnitsSold DESC, p.Title, limit input to 10 row(s) per chunk

-> Table scan on <temporary>

-> Aggregate using temporary table

-> Nested loop inner join (cost=11.5 rows=14)

-> Nested loop inner join (cost=6.55 rows=14)

-> Filter: (o.OrderDate >= <cache>((curdate() - interval 30 day))) (cost=1.65 rows=14)

-> Covering index scan on o using idx_orders_orderdate_orderid (cost=1.65 rows=14)

-> Index lookup on op using PRIMARY (OrderID = o.OrderID) (cost=0.257 rows=1)

-> Single-row index lookup on p using PRIMARY (ProductID = op.ProductID) (cost=0.257 rows=1)

‘

- All 14 orders were within the last 30 days, so the date filter matched every row.
- **EXPLAIN** before index: **full table scan**, 14 rows examined.
- **EXPLAIN** after index: **covering index scan**, still 14 rows examined.
- No improvement was visible because the filter could not exclude any rows.

Results with a dataset containing older orders

'-> Limit: 10 row(s)

-> Sort: TotalUnitsSold DESC, p.Title, limit input to 10 row(s) per chunk

-> Table scan on <temporary>

-> Aggregate using temporary table

-> Nested loop inner join (cost=7.46 rows=8)

-> Nested loop inner join (cost=4.66 rows=8)

-> Filter: (o.OrderDate >= <cache>((curdate() - interval 30 day))) (cost=1.86 rows=8)

-> Covering index range scan on o using idx_orders_orderdate_orderid over (''2025-11-04 00:00:00'' <= OrderDate) (cost=1.86 rows=8)

-> Index lookup on op using PRIMARY (OrderID = o.OrderID) (cost=0.263 rows=1)

-> Single-row index lookup on p using PRIMARY (ProductID = op.ProductID) (cost=0.263 rows=1)

‘

- Some orders fell outside the 30-day window.
- **EXPLAIN** showed a covering index range scan on the new index.
- Rows examined dropped from the full table (**14**) to only the recent orders (**≈ 8**).
- Joins on **Order_Product** and **Product** continued to use efficient primary-key lookups.