

Text2SQL AI Assistant

💡 Quick Start - Sample Questions

Popular queries to get you started:

📊 How many customers do we have?

🔍 Top 10 best-selling films?

🔍 Monthly rental trends

📊 Which category has highest revenue?

🔍 Customers with most rentals

🔍 Average rental duration

💬 Ask Your Question

Enter your question about the database:

Find the optimal inventory levels for each film per store

🚀 Generate Query & Analyze

Clear

Show SQL

Show Analysis

Visualization:

Auto-detect

⌚ AI Processing

📊 Results & Insights

✅ Query executed successfully!

✍️ Generated SQL Query

```
SELECT i.store_id, i.film_id, f.title AS film_title, COUNT(DISTINCT i.inventory_id) AS current_inventory_level, COUNT(r.rental_id) AS total_rentals
```

📝 Query Validation Notes

The user wants me to act as a Database Performance Specialist and review a given SQL query. The goal is to find the "optimal inventory levels for each film per store". The provided query calculates the total rentals and current inventory level for each film per store. I need to validate the query based on the provided checklist:

1. **Syntax correctness (especially WITH clause syntax):** The CTE syntax `WITH CTE_name AS (SELECT ...), CTE_name2 AS (SELECT ...)` is correct. The main query uses the CTEs correctly. Joins and selections seem syntactically correct.
2. **Performance optimization opportunities:**
 - o The query uses two separate CTEs, `FilmStoreRentalCounts` and `FilmStoreInventoryCounts`, which both group by `store_id` and `film_id`. These two counts could potentially be calculated in a single pass over the `inventory` and `rental` tables, or at least combined more efficiently.
 - o `FilmStoreRentalCounts` joins `rental` and `inventory` and groups. This is efficient for getting rental counts per inventory item, then aggregating by store/film.
 - o `FilmStoreInventoryCounts` groups `inventory` by store/film. This is efficient for getting inventory counts.
 - o The final join combines these two CTEs and the `film` table.

- Could we potentially calculate both counts in one CTE?
 - We could join `inventory` to `rental` (LEFT JOIN to include inventory items with no rentals).
 - Then group by `store_id` and `film_id`.
 - Inside the group, count the distinct `inventory_id` for the inventory level and count the `rental_id` for the rental count.
 - Let's try this approach.
 - `SELECT i.store_id, i.film_id, COUNT(DISTINCT i.inventory_id) AS current_inventory_level, COUNT(r.rental_id) AS total_rentals FROM inventory i LEFT JOIN rental r ON i.inventory_id = r.inventory_id GROUP BY i.store_id, i.film_id`
 - This single CTE calculates both metrics. This is likely more efficient than two separate CTEs and then joining them, especially if the intermediate results of the two original CTEs are large.
- The final query joins this single CTE with the `film` table. This seems efficient.
- The `ORDER BY` clause is reasonable for presenting the data.

3. Proper use of JOINs and relationships:

- `rental` JOIN `inventory`: Correctly links rentals to inventory items.
- `inventory` GROUP BY `store_id`, `film_id`: Correctly counts inventory per store/film.
- `FilmStoreInventoryCounts` LEFT JOIN `FilmStoreRentalCounts`: Correctly links inventory counts to rental counts. A LEFT JOIN from `FilmStoreInventoryCounts` is correct because we want to see all film/store combinations that have inventory, even if they have no rentals (which `COALESCE` handles).
- Joining with `film`: Correctly retrieves the film title.
- The proposed single-CTE approach uses `inventory` LEFT JOIN `rental` and then groups. This correctly links inventory items to their rentals (or lack thereof) before aggregating.

4. Appropriate use of indexes:

- The query involves joins and grouping on `inventory_id`, `store_id`, and `film_id` in the `inventory` table, and `inventory_id` and `rental_id` in the `rental` table.
- Indexes on `inventory(inventory_id)`, `inventory(store_id, film_id)`, `rental(rental_id)`, and `rental(inventory_id)` would be beneficial.
- An index on `film(film_id)` is also needed for the final join.
- The original query's CTEs would benefit from indexes on `inventory(store_id, film_id)` and `rental(inventory_id)`.
- The proposed single-CTE approach would benefit from indexes on `inventory(store_id, film_id, inventory_id)` (or separate indexes on `store_id`, `film_id`, `inventory_id`) and `rental(inventory_id)`.
- The final join benefits from indexes on `film(film_id)` and the CTE's output columns (`store_id`, `film_id`).
- The `ORDER BY` benefits from indexes on `store_id`, `total_rentals`, and `film_title`.
- The query itself doesn't specify indexes, but its performance relies on their existence. The structure of the query (especially the single-CTE version) is designed to leverage these indexes effectively.

5. Query safety (no dangerous operations):

The query is a `SELECT` statement. It does not perform any `INSERT`, `UPDATE`, `DELETE`, `DROP`, or other DDL/DML operations. It is safe.

6. Logic correctness for answering the question:

The question asks for "optimal inventory levels". The query provides `current_inventory_level` and `total_rentals_to_date`. While this data is essential for determining optimal levels (e.g., by analyzing rental velocity vs. stock), the query itself doesn't calculate or recommend an "optimal" level. It provides the raw data needed for such an analysis. The prompt asks me to "Review and optimize this SQL query for the question". The query provides data relevant to the question, but doesn't answer the question directly. However, the query as written correctly retrieves the current inventory and rental counts per film per store. Assuming the user intends to use this data to determine optimality elsewhere, the query's logic for retrieving this specific data is correct. The query's structure and output columns align with providing the necessary metrics.

Conclusion on Logic: The query correctly calculates the current inventory and total rentals per film per store. This data is a prerequisite for determining optimal inventory levels, even if the query doesn't perform the final "optimal" calculation itself. The query's logic for producing this specific dataset is sound.

Optimization Strategy: The most significant optimization opportunity is combining the two CTEs into one to reduce redundant grouping/processing.

Let's construct the optimized query using the single-CTE approach.

Original query:

```
WITH FilmStoreRentalCounts AS (
  SELECT i.store_id, i.film_id, COUNT(r.rental_id) AS total_rentals
  FROM rental r
  JOIN inventory i ON r.inventory_id = i.inventory_id
  GROUP BY i.store_id, i.film_id
),
FilmStoreInventoryCounts AS (
  SELECT i.store_id, i.film_id, COUNT(i.inventory_id) AS current_inventory_level
  FROM inventory i
  GROUP BY i.store_id, i.film_id
)
SELECT
  fisc.store_id,
  fisc.film_id,
  f.title AS film_title,
  fisc.current_inventory_level,
  COALESCE(fsdc.total_rentals, 0) AS total_rentals_to_date
FROM FilmStoreInventoryCounts fisc
JOIN film f ON fisc.film_id = f.film_id
LEFT JOIN FilmStoreRentalCounts fsdc ON fisc.store_id = fsdc.store_id AND fisc.film_id = fsdc.film_id
ORDER BY fisc.store_id, COALESCE(fsdc.total_rentals, 0) DESC, f.title;
```

Optimized query (single CTE).

```
WITH FilmStoreMetrics AS (
    SELECT
        i.store_id,
        i.film_id,
        COUNT(DISTINCT i.inventory_id) AS current_inventory_level, -- Count distinct inventory items per group
        COUNT(r.rental_id) AS total_rentals -- Count rentals linked to these inventory items
    FROM inventory i
    LEFT JOIN rental r ON i.inventory_id = r.inventory_id -- Use LEFT JOIN to include inventory items with no rentals
    GROUP BY i.store_id, i.film_id
)
SELECT
    fsm.store_id,
    fsm.film_id,
    f.title AS film_title,
    fsm.current_inventory_level,
    fsm.total_rentals AS total_rentals_to_date -- total_rentals will be 0 for items with no rentals due to LEFT JOIN and COUNT
FROM FilmStoreMetrics fsm
JOIN film f ON fsm.film_id = f.film_id
ORDER BY fsm.store_id, fsm.total_rentals DESC, f.title;
```

Let's double-check the `COUNT(DISTINCT i.inventory_id)` and `COUNT(r.rental_id)` logic in the single CTE. If an inventory item has multiple rentals, `LEFT JOIN rental r ON i.inventory_id = r.inventory_id` will produce multiple rows for that inventory item. `COUNT(DISTINCT i.inventory_id)` correctly counts each unique inventory item within the `store_id, film_id` group, regardless of how many rentals it has. `COUNT(r.rental_id)` correctly counts the number of rental records associated with *any* inventory item within the `store_id, film_id` group. If an inventory item has 3 rentals, and another has 2 rentals, and they are the only items for a given store/film, `COUNT(DISTINCT i.inventory_id)` will be 2, and `COUNT(r.rental_id)` will be 5. This is exactly what the original query achieved by joining the two CTEs. The `COALESCE` in the original query is handled naturally by `COUNT(r.rental_id)` in the single CTE, as `COUNT` returns 0 for groups where the joined table has no matching rows (due to the `LEFT JOIN`).

The optimized query using a single CTE is more efficient as it performs the join and aggregation in one pass over the relevant data, avoiding the overhead of creating and joining two intermediate result sets.

The original query is syntactically correct and logically sound for producing the required data points (current inventory and total rentals per film per store). However, it can be optimized.

Therefore, I should provide the optimized query. Thought: I have reviewed the original query based on the checklist.

1. **Syntax correctness:** The syntax, including the `WITH` clause, is correct.
2. **Performance optimization:** The use of two separate CTEs that are then joined can be optimized by combining the aggregation into a single CTE.
3. **Proper use of JOINs:** The original query uses JOINs correctly to link the CTEs and the film table. The proposed optimization uses a LEFT JOIN between inventory and rental before grouping, which is appropriate for counting rentals while retaining all inventory items.
4. **Appropriate use of indexes:** The query structure benefits from standard indexes on the joined and grouped columns. The optimized version is likely to utilize indexes more efficiently due to fewer intermediate steps.
5. **Query safety:** It's a `SELECT` query, so it's safe.
6. **Logic correctness:** The query correctly calculates the current inventory and total rentals per film per store, which are the necessary data points for the question, even if it doesn't perform the final "optimal" calculation. The logic for retrieving these specific metrics is sound.

The original query is syntactically correct and logically sound for its purpose (providing the data for analysis), but it can be optimized for performance. I will provide the optimized version. OPTIMIZED:

```
WITH FilmStoreMetrics AS (
    SELECT
        i.store_id,
        i.film_id,
        COUNT(DISTINCT i.inventory_id) AS current_inventory_level, -- Count distinct inventory items per group
        COUNT(r.rental_id) AS total_rentals -- Count rentals linked to these inventory items
    FROM inventory i
    LEFT JOIN rental r ON i.inventory_id = r.inventory_id -- Use LEFT JOIN to include inventory items with no rentals
    GROUP BY i.store_id, i.film_id
)
SELECT
    fsm.store_id,
    fsm.film_id,
    f.title AS film_title,
    fsm.current_inventory_level,
    fsm.total_rentals AS total_rentals_to_date -- total_rentals will be 0 for items with no rentals due to LEFT JOIN and COUNT
FROM FilmStoreMetrics fsm
JOIN film f ON fsm.film_id = f.film_id
ORDER BY fsm.store_id, fsm.total_rentals DESC, f.title;
```

1521

ROWS

5

COLUMNS

0.05s

EXEC TIME

Data Visualization

film_id vs store_id



Try other visualizations:

film_id vs store_id

store_id by film_title

AI Insights & Analysis

Here is an analysis of the provided data regarding inventory levels and rental performance:

1. Key Findings and Insights:

- The dataset provides a snapshot of current inventory levels (`current_inventory_level`) against cumulative historical demand (`total_rentals_to_date`) for each film at specific stores.
- Optimal inventory levels cannot be definitively calculated from this data alone, as it lacks information on rental velocity over specific periods (e.g., weekly/monthly rentals), lead times for restocking, storage costs, and stockout costs.
- However, the data allows us to identify films and stores where current inventory levels appear potentially misaligned with historical demand, serving as a starting point for inventory optimization efforts.

2. Notable Patterns or Trends:

- We would expect, ideally, that films with higher `total_rentals_to_date` would have correspondingly higher `current_inventory_level` to meet ongoing demand, and vice versa. Analyzing the full dataset would reveal if this general trend holds or if there are significant deviations.
- Differences in rental performance for the same film across different stores can indicate varying local demand, suggesting that optimal inventory levels are store-specific, not just film-specific.
- Some films might show consistently high or low rental performance across all stores, indicating overall popularity or lack thereof.

3. Business Implications:

- Understocking:** Films with high historical rentals but low current inventory risk stockouts, leading to lost rental revenue and potential customer dissatisfaction.
- Overstocking:** Films with low historical rentals but high current inventory incur unnecessary holding costs (storage space, potential damage, obsolescence) and tie up capital that could be used elsewhere.
- Optimizing inventory based on demand patterns can improve profitability by reducing costs and maximizing revenue opportunities.
- Understanding store-specific demand allows for tailored inventory strategies rather than a one-size-fits-all approach.

4. Recommendations for Data Visualization:

- Scatter Plot:** Plot `total_rentals_to_date` on the x-axis and `current_inventory_level` on the y-axis. Each point represents a film-store combination. This visualization helps identify outliers: points far above the general trend (potentially overstocked) or far below (potentially understocked). Color-coding points by `store_id` or `film_id` (or genre, if available) can reveal further patterns.
- Bar Chart (Grouped/Stacked):** For a specific store or a selection of films, show bars representing `total_rentals_to_date` and `current_inventory_level` side-by-side for each film. This allows for direct comparison of demand vs. stock for individual items.
- Heatmap:** A heatmap showing `total_rentals_to_date` (or a ratio like rentals per inventory item) for each film across each store. This can quickly highlight which films are popular/unpopular in which locations.

5. Anomalies or Interesting Observations:

- Look for instances where `current_inventory_level` is very high (e.g., in the top 10% of rentals) but `total_rentals_to_date` is very low (e.g., 1 or 2). These are strong candidates for potential understocking.

- LOOK for instances where `total_rentals_to_date` is very high (e.g., in the top 10% of rentals) but `current_inventory_level` is very low (e.g., 1 or 2). These are strong candidates for being understocked.
- Conversely, identify films with very low `total_rentals_to_date` (e.g., in the bottom 10%) but relatively high `current_inventory_level` (e.g., 3 or more). These may be overstocked.
- Investigate films with 0 rentals but non-zero inventory (if any exist in the full dataset) - are these new releases, or films with no demand?
- Are there films with identical `film_id` but vastly different rental performance or inventory levels across stores? This would reinforce the need for store-level optimization.

[View Raw Data](#)

	store_id	film_id	film_title	current_inventory_level	total_rentals_to_date
0	1	535	LOVE SUICIDES	4	20
1	1	55	BARBARELLA STREETCAR	4	18
2	1	489	JUGGLER HARDLY	4	18
3	1	545	MADNESS ATTACKS	4	18
4	1	603	MOVIE SHAKESPEARE	4	18
5	1	938	VELVET TERMINATOR	4	18
6	1	941	VIDEOTAPE ARSENIC	4	18
7	1	971	WHALE DIVING	4	18

[Download CSV](#) [Download JSON](#)