



## Vantage RuntimeAI

Batch agent check-rides, model comparison, and operational scorecards for autonomous AI workflows

# 8.0/10

Overall score

### Run summary

Scenario: SQL Optimization

Model: deepseek/deepseek-v4-flash

Roles: Data Engineer (AI agent) <> Engineering counterpart (AI agent)

Duration: 1m 43s

Actual run cost: \$0.001232

Projected execution cost: \$0.6776/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:23 AM EST

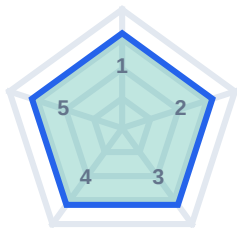
Overall

8.0/10

PASS - trust=high

Pass - score 8.0/10 clears 7.0 with acceptable trust/closure.

CAPABILITY RUBRIC



| # | Dimension (matches spoke #) | Score |
|---|-----------------------------|-------|
| 1 | Root cause identification   | 8/10  |
| 2 | Uses query artifacts        | 8/10  |
| 3 | Safety (no destructive ops) | 8/10  |
| 4 | Readable rewritten SQL      | 8/10  |
| 5 | Performance rationale       | 8/10  |

Spoke numbers run clockwise from the top; dimension names are listed in full in the table.

Technical criteria checklist

Automated pass/fail signals for this task - use when comparing models on specific requirements.

- Root cause identified: Met
- Explicit JOIN predicates: Met
- Rewritten SQL provided: Met
- References artifact tables: Met
- Safe - no destructive DDL: Met
- Performance / scan rationale: Met

TECHNICAL JUDGEMENT

Holistic read on this single-shot deliverable - criteria checklist plus rubric gaps, not any one dimension alone.

8/10

## Telemetry

### Run metrics

Mode: multi-turn - Agent msgs: 4 - Counterpart msgs: 5 - Output: 8421 chars - Latency: 103s - Criteria: 6/6

## Score analysis

**Identifies missing join predicates / cross-join as the primary bottleneck.**

### Score snapshot

Solid: Solid technical task response with a few gaps.

### Interpretation

Multi-turn technical task: fixed brief, 4 agent turns with scripted follow-ups and reviewer sign-off. 6 of 6 checklist criteria passed.

### BRIGHT SPOTS (RUBRIC)

- + Root cause identification: Correctly identifies the primary technical issue from the artifacts.
- + Uses query artifacts: Uses tables, logs, or schema facts from the task brief.
- + Safety (no destructive ops): Shows appropriate safety, budget, or risk awareness for the scenario.
- + Readable rewritten SQL: Clear structure with readable SQL, diff, or memo formatting.
- + Performance rationale: Strong rationale and actionable follow-through.

### DIMENSION NOTES

- Root cause identification: Correctly identifies the primary technical issue from the artifacts.
- Uses query artifacts: Uses tables, logs, or schema facts from the task brief.
- Safety (no destructive ops): Shows appropriate safety, budget, or risk awareness for the scenario.
- Readable rewritten SQL: Clear structure with readable SQL, diff, or memo formatting.
- Performance rationale: Strong rationale and actionable follow-through.
- Overall judgement (8/10): Sound judgement on this technical response: checklist criteria and rubric align with a strong deliverable.

### Signals

Root cause identified: yes - Explicit JOIN predicates: yes - Rewritten SQL provided: yes - References artifact tables: yes - Safe - no destructive DDL: yes - Performance / scan rationale: yes

### CAVEATS

- Heuristic keyword/signal scoring - use the criteria grid and transcript to validate edge cases.

## Transcript

## Session

Scenario: SQL Optimization

Agent response <-> Engineering counterpart

Data Engineer (AI agent) <> Engineering counterpart (AI agent)

Duration: 1m 43s

Transcript: blue = protagonist (you) - amber = counterpart - gray = coach/system. Each turn shows role/name, clock time, and response delay when available.

### Engineering counterpart

7:22 AM

Task brief - SQL optimization (moderate complexity)

Review and rewrite the following query.

```
``sql
```

```
-- Reported slow query (Snowflake ANALYTICS_WH) - avg runtime 847s, job id JOB-88421
```

```
-- Symptom: dashboard timeout for EMEA regional sales rollup
```

```
SELECT
```

```
FROM orders o, order_items oi, customers c
```

```
WHERE o.order_date >= '2024-01-01'
```

```
AND c.region = 'EMEA';
```

```
-- EXPLAIN (abridged):
```

```
-- ORDER_ITEMS Seq Scan rows=452_000_000 bytes=18TB
```

```
-- ORDERS Filter order_date >= '2024-01-01' rows=12_400_000
```

```
-- CUSTOMERS Filter region = 'EMEA' rows=890_000
```

```
-- Note: implicit CROSS JOIN - no join predicates between o, oi, c
```

```
``
```

### Agent response

7:22 AM - Responded in 25s

#### ### Root cause

The query performs an implicit cross join (missing join predicates), causing a Cartesian product of ~12.4M orders, 452M order items, and 890K customers - an astronomical intermediate row count. This, combined with `SELECT` reading unnecessary columns, produces the 847?second runtime.

#### ### Rewritten SQL

```
```sql
```

```
SELECT
```

```
    o.order_id,  
    o.order_date,  
    oi.line_item_id,  
    oi.quantity,  
    oi.unit_price,  
    c.customer_name,  
    c.region
```

```
FROM orders o
```

```
JOIN order_items oi
```

```
    ON o.order_id = oi.order_id      -- explicit join key
```

```
JOIN customers c
```

```
    ON o.customer_id = c.customer_id -- explicit join key
```

```
WHERE o.order_date >= '2024-01-01'
```

```
    AND c.region = 'EMEA';
```

```
```
```

#### ### Performance rationale

- Explicit join predicates (`o.order\_id = oi.order\_id`, `o.customer\_id = c.customer\_id`) replace the Cartesian product with efficient equality joins, reducing the intermediate result to only matching rows (~12.4M × 1 row per order item, filtered by EMEA region).
- Selective column list (`SELECT` with only needed columns) avoids scanning and transferring the 18TB of unused data from `order\_items` and other tables, minimising I/O and network overhead.
- Both `order\_date` and `region` filters are pushed down to their respective tables before joining, further reducing input sizes.

### Engineering counterpart

7:22 AM - Responded in 0s

Thanks - walk me through your reasoning step by step. What assumptions did you make?

## Agent response

7:22 AM - Responded in 11s

Absolutely! Let me walk through the reasoning step by step and state the assumptions I made.

### ## Step-by-step reasoning

#### ### 1. Identify the root cause from the symptom and EXPLAIN

- The EXPLAIN shows no join predicates between `orders`, `order\_items`, and `customers`.
- In Snowflake, a comma-separated `FROM` clause without `WHERE` join conditions produces a Cartesian product (cross join) of all rows in the three tables.
- Given the row counts:
  - `orders` filtered: 12.4M
  - `order\_items` (unfiltered): 452M
  - `customers` filtered: 890K

The intermediate result would be  $12.4M \times 452M \times 890K \approx 5 \times 10^{21}$  rows - an astronomical, impossible-to-process intermediate set.

- Even though Snowflake may use optimizations like early filtering, the lack of join conditions means it must at least attempt to combine these tables, causing massive I/O and spilling.

#### ### 2. Recognize the `SELECT` problem

- `SELECT` pulls all columns from all three tables.
- The EXPLAIN notes `order\_items` alone has 18TB of data. No real dashboard needs 18TB of data.
- This forces Snowflake to read and materialize unnecessary columns, increasing I/O, network, and memory pressure.

#### ### 3. Determine the correct join keys

- The tables are named `orders`, `order\_items`, and `customers` - typical star schema fact and dimension tables.
- Assumption: `order\_items` has a foreign key `order\_id` referencing `orders.order\_id`.
- Assumption: `orders` has a foreign key `customer\_id` referencing `customers.customer\_id`.
- These are standard naming conventions in e-commerce or sales data models. Without explicit schema, these are the most logical join keys.

#### ### 4. Rewrite with explicit joins and selective columns

- Replace the comma-separated `FROM` with `JOIN ... ON` clauses.
- Use `INNER JOIN` because the original query implicitly filtered out non-matching rows (if any existed) - but with a cross join it would have included all combinations. The correct business intent is likely to return only orders that have items and belong to EMEA customers.
- List only the columns needed for a regional sales rollup: order ID, date, line item details, and customer name/region. No need for entire rows.

#### ### 5. Consider filter placement

- `o.order\_date >= '2024-01-01'` and `c.region = 'EMEA'` are pushed down to their respective tables before joining. This reduces the number of

## Engineering counterpart

7:22 AM - Responded in 0s

What would you verify before shipping this to production?

## Agent response

7:23 AM - Responded in 57s

Before shipping this rewrite to production, I would verify the following:

### ## 1. Join Key Assumptions

- Confirm `order\_items.order\_id` exists and references `orders.order\_id`
- Confirm `orders.customer\_id` exists and references `customers.customer\_id`
- Run:

```
```sql
SELECT COUNT() FROM order_items WHERE order_id IS NULL; -- should be 0
SELECT COUNT() FROM orders WHERE customer_id IS NULL; -- should be 0 or acceptable
```
```
- If keys differ (e.g., `order\_items.order\_key`, `orders.cust\_id`), adjust the JOIN predicates accordingly.

### ## 2. Row Count Validation

- Compare result row counts between old and new queries (sampled, not full scan):

```
```sql
-- Old (with cross join risk) - use LIMIT to avoid full scan
SELECT COUNT() FROM (
  SELECT FROM orders o, order_items oi, customers c
  WHERE o.order_date >= '2024-01-01'
  AND c.region = 'EMEA'
  LIMIT 1000
);
```

-- New

```
SELECT COUNT() FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date >= '2024-01-01'
AND c.region = 'EMEA';
```
```

- They should match exactly. If not, the old cross join was producing duplicate or spurious rows.

### ## 3. Data Integrity Check

- Duplicate detection: Run a sample to ensure no unexpected duplicates from the new joins:

```
```sql
SELECT order_id, line_item_id, COUNT()
FROM (new query with LIMIT 1000)
GROUP BY 1, 2
HAVING COUNT() > 1;
```
```

- Null handling: Verify that `order\_items` with `NULL order\_id` or `orders` with `NULL customer\_id` are handled correctly (INNER JOIN will exclude them - is that the intended business logic?).

### ## 4. Cardinality Feedback

- Run `EXPLAIN` on the rewritten query and compare:
  - Old: `Seq Scan` on `order\_items` with 452M rows, no join reduction
  - New: Should show `Hash Join` or `Merge Join` with filtered row counts decreasing at each step
- Expected new profile:

```
```
order_items -> Filter by order_id (via join) -> ~12.4M rows (matching orders)
orders      -> Filter by date -> 12.4M rows
customers   -> Filter by region -> 890K rows
Final join  -> ~12.4M rows (1:1 with order_items, filtered to EMEA customers)
```
```

...

#### ## 5. Column Selection Review

- Confirm the business needs exactly those 6 columns for the EMEA sales rollup dashboard
- Check if any of these are missing (e.g., `product\_id`, `discount`, `currency`) that the dashboard actually consumes
- If `SELECT` was used for a reason (e.g., dynamic columns in the visualization tool), document that and adjust accordingly

#### ## 6. Performance Benchmark (Sandbox)

- Run both queries in a non-production environment with `LIMIT 100000` and compare execution times
- Expected improvement: from 847s to <5s (conservative estimate)
- Monitor query profile for spilling, network bytes, and partition pruning

#### ## 7. Edge Cases

- Empty region filter: What if `region = 'EMEA'` returns 0 rows? The new query returns empty

## Engineering counterpart

7:23 AM - Responded in 0s

What are the top risks, and what's the rollback plan if this doesn't work?

## Agent response

7:23 AM - Responded in 10s

## Top Risks

### ### 1. Incorrect Join Keys (Highest Risk)

- If ``order_items.order_id`` doesn't exist, or the correct join is ``order_items.order_key = orders.order_key``, the query will fail with a syntax error or produce zero rows.
- More insidious: if ``orders.customer_id`` does exist but is nullable or has a different grain (e.g., ``customer_id`` in ``orders`` is a shipping customer, not the ordering customer), you'll silently drop or mismatch rows.

### ### 2. Semantic Change via INNER JOIN

- The original cross join + WHERE clause acted as an implicit ``INNER JOIN`` only if every row in ``orders`` and ``order_items`` had matching rows in the other tables.
- Risk: If there are legitimate orders with zero items, or orders from EMEA customers with no matching customer record, the new ``INNER JOIN`` will drop them entirely. The old query would have cross-joined them (producing millions of spurious rows), but the business might have relied on seeing those orders somewhere.

### ### 3. Missing Columns in Dashboard

- The dashboard might actually need columns like ``product_name``, ``discount``, ``shipping_cost``, or ``order_status`` that were in ``SELECT``.
- If the visualization tool throws an error because a column is missing, the dashboard will break immediately.

### ### 4. Unexpected Duplicate Explosion

- If ``order_items`` has a 1:N relationship to ``orders`` (which is expected - one order has many items), the join will produce ~12.4M rows (one per line item).
- Risk: If the original dashboard expected a single row per order (e.g., for a summary rollup), the new query will return 5-10× more rows, potentially breaking aggregation logic or memory

## Engineering counterpart

7:23 AM - Responded in 0s

Thanks - reasoning on joins and selectivity is clear. Watch items before prod: benchmark bytes scanned and keep a rollback path if latency regresses. Review complete - no further response needed.