



## 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: amazon/nova-micro-v1

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

Duration: 9s

Actual run cost: \$0.000406

Projected execution cost: \$0.2233/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:22 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: 10677 chars - Latency: 9s - 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: 9s

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

### Engineering counterpart

7:21 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

7:21 AM - Responded in 1s

Certainly! Here's a structured review and rewrite of the problematic SQL query.

#### ### 1) Root Cause

The query is slow because it's performing a Cartesian product (`CROSS JOIN`) between three large tables: `orders`, `order\_items`, and `customers` without any explicit JOIN conditions. This leads to an enormous and unnecessary data set being processed, resulting in significant runtime.

#### ### 2) Rewritten SQL with explicit JOIN predicates and selective columns

```
```sql
SELECT
  o.order_id,
  o.order_date,
  oi.item_id,
  oi.order_id AS order_item_order_id,
  c.customer_id,
  c.region
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';
```
```

#### ### 3) Brief Performance Rationale

By adding explicit JOIN predicates, we restrict the join operation to only rows that have a matching relationship between `orders`, `order\_items`, and `customers`. This will drastically reduce the number of rows processed in the `JOIN` operations. Additionally, selecting only necessary columns (`order\_id`, `order\_date`, `item\_id`, `order\_id` from `order\_items`, `customer\_id`, `region` from `customers`) eliminates the overhead of transferring and processing unnecessary data. These changes will result in far fewer rows being scanned and processed, thus improving the query performance significantly.

### Engineering counterpart

7:21 AM - Responded in 0s

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

## Agent response

7:21 AM - Responded in 2s

Certainly! Let's walk through the reasoning step by step, including assumptions made in the process:

### ### Step-by-Step Reasoning:

1. Identify the root cause of the performance issue:

- The query performs a Cartesian product (cross join) among `orders`, `order\_items`, and `customers` without any explicit JOIN conditions. This leads to substantial data bloat and inefficiency.

2. Look at the reported execution plan:

- `ORDER\_ITEMS Seq Scan`: Scanning 452 million rows and 18 TB.
- `ORDERS Filter` order\_date >= '2024-01-01': Filtering down to 12.4 million rows.
- `CUSTOMERS Filter` region = 'EMEA': Filtering down to 890 thousand rows.

These implicit Cartesian products between massive and filtered data sets result in considerable overhead and runtime.

3. Recognize the implicit behavior:

- The original query lacks explicit JOIN conditions, so Snowflake must implicitly perform joins which is akin to doing a Cartesian product.

4. Apply explicit JOIN conditions:

- The join keys are likely `order\_id` between `orders` and `order\_items`, and `customer\_id` between `orders` and `customers`.
- This transforms the Cartesian product into explicit, necessary joins.

5. Select selective columns:

- Instead of using `SELECT`, we only select columns that are essential for the query's purpose (e.g., `order\_id`, `order\_date`, `item\_id`, `customer\_id`, `region`). This avoids loading unnecessary data.

### ### Assumptions:

1. Common keys assumption:

- Assume there are columns (`order\_id` in `orders` and `order\_items`, `customer\_id` in `orders` and `customers`) that define the join relationships between the tables.

2. Filter conditions assumption:

- Assume that the `WHERE` clauses filter down the `orders` ('by order\_date') and `customers` ('by region') effectively without needing a join with `order\_items` initially.

3. Logical Join sequence assumption:

- Assume that joining `orders` with `order\_items` first, then joining the result with `customers` makes sense based on the filtering criteria applied.

### ### Final Rewritten Query:

```
```sql
SELECT
  o.order_id,
  o.order_date,
  oi.item_id,
  oi.order_id AS order_item_order_id,
  c.customer_id,
  c.region
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';
'''
```

#### ### Performance Rationale:

##### 1. Explicit JOINS instead of Implicit Cartesian Product:

- Instead of a Cartesian product which could multiply millions of rows, explicit JOINS only combine rows when the join conditions are satisfied.

##### 2. Selective Column Usage:

- Reducing the number of columns fetched by avoiding `SELECT \*` minimizes data transfer and processing time.

By addressing the primary cause of the performance issue (i.e., the lack of explicit JOIN predicates and the use of `SELECT \*`), this rewritten query is expected to significantly improve runtime and efficiency.

## Engineering counterpart

7:21 AM - Responded in 0s

What would you verify before shipping this to production?

### Agent response

7:22 AM - Responded in 2s

Before shipping this query optimization to production, you should verify the following:

#### ### 1. Schema and Data Integrity:

- Table Structures: Ensure the `orders`, `order\_items`, and `customers` tables have the columns `order\_id`, `customer\_id`, `item\_id`, `region`, and `order\_date`.
- Data Consistency: Check if the values in `order\_id`, `customer\_id`, and `order\_date` columns are consistent with what's expected (i.e., `order\_id` in `orders` table matches `order\_id` in `order\_items`, and `customer\_id` links customers to orders).

#### ### 2. Performance Testing:

- Execution Plan: Run the new query and utilize Snowflake's `EXPLAIN` statement to understand the execution plan and optimize further if necessary.
- Benchmarking: Measure the query performance with a representative dataset to ensure it runs efficiently in production.

#### ### 3. Edge Cases and Data Volume:

- Date Ranges: Verify the performance for various date ranges, including minimal and maximal spans.
- Full vs Partial Data: Test with a full dataset to make sure the query performs well across all potential data sizes.

#### ### 4. Performance Metrics:

- Execution Time: Ensure the execution time is significantly reduced compared to the original query.
- Row Count: Confirm the row count output makes logical sense based on the filters and joins applied.

#### ### 5. Dependency and Blocking Issues:

- Dependencies: Ensure no other processes depend on the old query and that the rewrite does not introduce conflicts.
- Concurrency: Check if there are any concurrency issues that might affect this new query in the context of other operations on the same tables.

#### ### 6. Testing in Staging Environment:

- Staging Data: Use a staging environment that mirrors production for testing the new query.
- Regression Testing: Perform comprehensive tests to ensure no existing functionality is broken.

#### ### 7. Logging and Monitoring:

- Logging: Add appropriate logging to capture any runtime issues or anomalies when the query is deployed.
- Monitoring: Set up monitoring to track the performance and any changes in query runtime post-deployment.

#### ### 8. Rollback Plan:

- Backup: Have a rollback plan in place, including a backup of the existing query and data, in case the new query fails.

#### ### 9. Documentation and Communication:

- Documentation: Document the changes made, why they were necessary, and how they impact the overall system.
- Communication: Inform and get buy-in from stakeholders, including developers, data engineers, and end-users.

By thoroughly verifying these points, you can ensure that the revised query is robust, performant, and ready for production deployment.

**Engineering counterpart**  
**7:22 AM - Responded in 0s**

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

## Agent response

7:22 AM - Responded in 3s

When deploying a new query in a production environment, it's important to be prepared for potential issues and have a rollback plan. Here's a breakdown of top risks associated with query optimizations and a potential rollback plan:

### ### Top Risks

#### 1. Performance Issues:

- Incorrect Execution Plan: The new query might not execute as anticipated, leading to poor performance due to misinterpretation of join conditions or lack of proper indexing.
- Increased Latency: Even if the query runs faster in test situations, it might introduce latency under production load conditions due to changes in data distribution.

#### 2. Data Accuracy:

- Incorrect Results: If the query logic is incorrect, it could return inaccurate data or miss pertinent rows, leading to incorrect business decisions.

#### 3. System Dependencies:

- Downtime for Interdependent Services: Other applications dependent on the results from this query might experience downtime or errors if the changes result in unexpected outcomes.

#### 4. Maintenance and Monitor Setup:

- Failure to Monitor Properly: Inadequate monitoring could result in unrecognized issues that go unresolved until they become significant problems.

#### 5. Compatibility Issues:

- Interference with Other Queries: The change might interact poorly with other concurrent queries, potentially causing locking or contention issues.

### ### Rollback Plan

To mitigate the risks and ensure a smooth transition, it's critical to have a detailed rollback plan:

#### 1. Backup:

- Data Backup: Before making the change, ensure that the production database has a recent backup.
- Old Query Version: Create a versioned backup of the old query, so it can be re-deployed if necessary.

#### 2. Staging Testing:

- Thoroughly test the new query in a staging environment that mirrors production. Utilize this environment to run various tests including stress testing and performance evaluations under different conditions.

#### 3. Dry Run:

- Conduct a "dry run" where the new query is deployed to a small percentage of live traffic or in a shadow mode. This minimizes exposure while still getting real-world data.

#### 4. Monitoring and Alerts:

- Implement real-time monitoring and set up alerts for key performance indicators (KPIs) such as execution time, response time, and data integrity.
- Monitor for any anomalies or performance degradation on critical business processes dependent on this query.

#### 5. Rapid Rollback Procedures:

- Develop a clear, step-by-step process for redeploying the old query if needed. This should involve:
  - Automatically restoring data backups.
  - Reversing any schema changes or deployment changes made.
  - Re-enabling the old query and disabling the new one if it fails.

6. Communication:

- Inform all relevant stakeholders (development, operations, business units) about the rollback plan and the signs or conditions under which it would be enacted.

7. Post-Deployment Review:

- After the deployment stabilizes, conduct a review to assess what worked, what didn't, and how future changes can be better managed and mitigated.

By comprehensively addressing these risks with a detailed rollback plan, you greatly reduce the potential negative impact of deploying an optimized query, ensuring a safer transition into production.

**Engineering counterpart****7:22 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.