Optimizing Queries Using Materialized Views: A Practical, Scalable Solution

Microsoft Research

4th March, 2025

Presented by: Yash Bhisikar

Prereqs: Views

```sql
CREATE VIEW SalesSummary2025 AS
SELECT
    p.product_name,
    c.category_name,
    SUM(od.quantity) AS total_units_sold,
    SUM(od.quantity * p.unit_price) AS total_revenue,
    COUNT(DISTINCT o.order_id) AS total_orders
FROM
    orders o
JOIN order_details od ON o.order_id = od.order_id
JOIN products p ON od.product_id = p.product_id
JOIN categories c ON p.category_id = c.category_id
WHERE
    o.order_date BETWEEN '2025-01-01' AND '2025-03-04'
GROUP BY
    p.product_name, c.category_name
```

```sql
CREATE MATERIALIZED VIEW DailySalesCache
REFRESH EVERY 1 HOUR
AS
SELECT
    order_date,
    SUM(quantity * unit_price) AS daily_total
FROM orders
JOIN order_details USING (order_id
GROUP BY order_date;
```

```sql
SELECT * FROM SalesSummary2025
ORDER BY total_revenue DESC
LIMIT 10;
```

Examples of creating a view and querying it

Problem Statement

Given a relational expression in SPJG form, find all materialized (SPJG) views from which the expression can be computed and, for each view found, construct a substitute expression equivalent to the given expression.

Keywords: (SPJG == Select, Project, Join, GroupBy)

Key Contributions:

1. **A fast view-matching algorithm:** identifying which materialized views can be used to “answer” a query
2. **Scalability:** A filter tree index to quickly prune away the views irrelevant to the optimization, performant with 1000+ views
3. **Integration with cost-based optimizers:** Allowing query rewrites (on SQL Server) to compete in the normal optimization process

When is a query expression computable from a view?

1. The view contains all rows needed by the query expression
2. All required rows can be selected from the view
3. All output expressions can be computed from the output of the view
4. All output rows occur with the correct duplication factor

Column equivalence classes - I

1. Let's say we have a selection predicate: $W = P_1 \wedge P_2 \wedge P_3 \ldots \wedge P_n$
2. We split the conjuncts as $W = PE \wedge PNE$
    a. PE contains column equality predicates, $T_i.C_p = T_j.C_q$ (T_i & T_j are tables, C_p and C_q are corresponding column references)
    b. PNE contains the remaining predicates
3. After the *PE* predicates are applied, we will get **"connected components"**, or ***equivalence classes***, of columns. These can be interchangeably used in *PNE* predicates and output expressions.

Column equivalence classes - II

Advantages:

1. Can reroute column references within the query/view
2. Use an index on any one of the columns from a class to avoid redundant checks

```sql
SELECT *
FROM employees
JOIN salaries ON employees.id = salaries.employee_id
WHERE employees.id = 100;
```

```sql
SELECT *
FROM students
JOIN grades ON students.student_id = grades.student_id
JOIN attendance ON grades.student_id = attendance.student_id
```

Do all the required rows exist in the view?

Tackling the *where* clause

1. Given: $T_1, T_2, T_3 \ldots T_m$ are the tables. W_q is the where clause in the query, whereas W_v is the *where* clause in the view
2. The view contains ***all*** the rows that the query needs if: $W_q \Rightarrow W_v$ (logical implication)
3. Write $W_q = P_{q,1} \wedge P_{q,2} \wedge \ldots \wedge P_{q,m}$ and $W_v = P_{v,1} \wedge P_{v,2} \wedge \ldots \wedge P_{v,n}$

   We need a quick way to decide whether the implication holds

Deciding whether the implication holds - I

- **Naive way:** check that every conjunct $P_{v,i}$ in W_v matches a conjunct $P_{v,i}$ in W_q
  a. Devil is in the details: Syntactic match, String comparison, Commutativity/Transitivity
  b. Eg: $(\frac{A}{2} + \frac{B}{2}) \times 10 = B \times 2 + A \times 5$
  c. This ensures completeness, but comes at the cost of sophistication and computation time
- Do we really need complete correctness?
- False Negatives (deciding that view cannot be used for the query, when it actually can be) are still preferable over False Positives(deciding that view can be used for the query, when it actually cannot be)
- The trick: We trade off completeness for speed

Deciding whether the implication holds - II

1. Rewrite the implication as $PE_q \wedge PR_q \wedge PU_q \Rightarrow PE_v \wedge PR_v \wedge PU_v$
   Where W_q is split up as follows (W_v is split up similarly) :
   a. PE_q consists of all column equality predicates $(T_i.C_p = T_j.C_r)$,
   b. PR_q contains range predicates $(T_i.C_p\ op\ c)$, and
   c. PU_q is the residual predicate.

2. Using rules of boolean algebra, we rewrite it as:

$$(PE_q \wedge PR_q \wedge PU_q \Rightarrow PE_v) \wedge (PE_q \wedge PR_q \wedge PU_q \Rightarrow PR_v) \wedge (PE_q \wedge PR_q \wedge PU_q \Rightarrow PU_v)$$

Deciding whether the implication holds - III

- Observe that we can drop some of the antecedents to make the tests stronger $(A \Rightarrow C) \Rightarrow (AB \Rightarrow C)$, and not vice-versa
  1. $PE_q \Rightarrow PE_v$ (Equijoin subsumption test)
  2. $PE_q \land PR_q \Rightarrow PR_v$ (Range subsumption test)
  3. $PE_q \land PU_q \Rightarrow PU_v$ (Residual subsumption test)
- These tests are stronger than their original versions, but we will miss out on different cases
  Eg: by dropping PR_q from the antecedent of the equijoin test we will miss cases when the query equates two columns to the same constant, say, $(A = 2) \land (B = 2)$ and the view contains the weaker predicate $(A = B)$
- Similar problems with the other tests

Summary of the three tests

Equijoin Subsumption Test

Goal: Ensure the view's join conditions are compatible with the query's.

How: Check if every column equivalence in the view (e.g., A = B) is also enforced by the query, either directly or transitively.

What it assures: The view doesn't miss any equality constraints required by the query.

Example: If the view equates $A = B$ and $B = C$, the query must imply the same (e.g., via $A = C$).

Range Subsumption Test

Goal: Ensure the view's range predicates (e.g., col > 100) are at least as permissive as the query's.

How: Compare numeric/date ranges (e.g., view has col ≥ 50, query needs col ≥ 100). The view's range must fully cover the query's range.

What it assures: The view doesn't exclude rows the query requires.

Example: If the query filters price BETWEEN 150 AND 200, the view must include this range (e.g., price ≥ 100).

Residual Subsumption Test

Goal: Ensure the view's non-join, non-range predicates (e.g., LIKE clauses) match the query's.

How: Check if every residual predicate in the view (e.g., p_name LIKE '%steel%') exists in the query.

What it assures: The view doesn't filter out rows the query needs due to stricter conditions.

Example: If the view has p_name LIKE '%steel%, the query must include this predicate.

Some other tests …

1. Can the required rows be selected?
2. Can output expressions be computed?
3. Do rows occur with correct duplication factor?

And a worked out example in the paper!

Views With Extra Tables

Core approach: identify cardinality preserving joins with the non-null foreign key constraints

Steps:

1. **Foreign-Key Join Graph**: A directed graph is built where edges represent joins between tables that meet strict criteria (equijoin on non-null foreign keys referencing unique keys).
2. **Elimination of Extra Tables**: The algorithm iteratively removes extra tables by checking if they can be "pruned" via these validated joins. If all extra tables are eliminated through this process, the view is deemed compatible

Fast Filtering of Views - The Filter Index Tree - I

1. in-memory index structure designed to efficiently narrow down candidate materialized views
2. Structure:
    a. **Multiway search tree** with leaves at the same level
    b. Nodes contain (key, pointer) pairs where keys are **sets of values**
    c. Uses **lattice indexes** internally to organize keys via subset/superset relationships
3. Partitioning Conditions
    a. **Source Tables**: Requires view tables ⊆ query tables (via lattice index on table sets)
    b. **Hub Condition**: Checks if view's minimal table set (after eliminating extra tables via FK joins) ⊆ query tables
    c. **Output Columns**: Verifies query columns exist in view's output (expanded via equivalence classes)
    d. **Grouping Columns**: Ensures query grouping columns ⊆ view grouping columns (with FD awareness)
    e. **Residual Predicates**: Requires view's non-join/range predicates ⊆ query predicates
    f. **Range Constraints**: Checks view ranges contain query ranges (via equivalence class ranges)

Fast Filtering of Views - The Filter Index Tree - II

Key Optimizations in a nutshell

- Lattice indexes enable fast subset/superset checks without linear scans
- Compares text signatures for expressions/predicates (with column equivalence awareness)
- Progressively prunes candidates across multiple constraint dimensions(partition conditions)

Results - I

Figure 2: Optimization time as a function of the number of views.

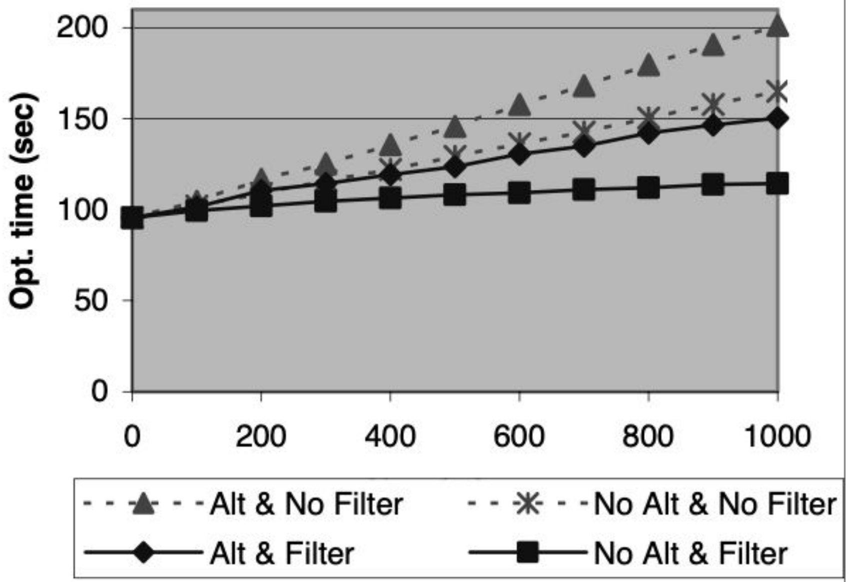
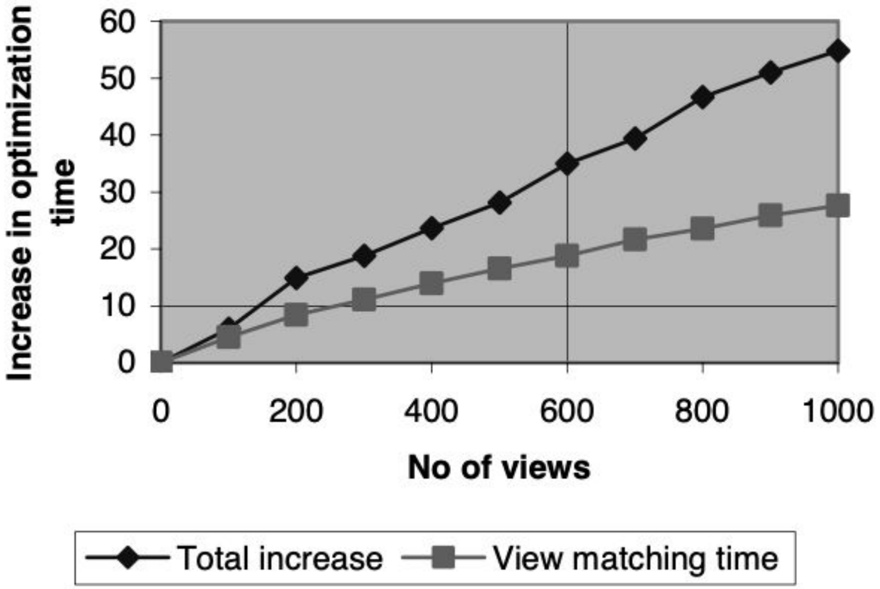
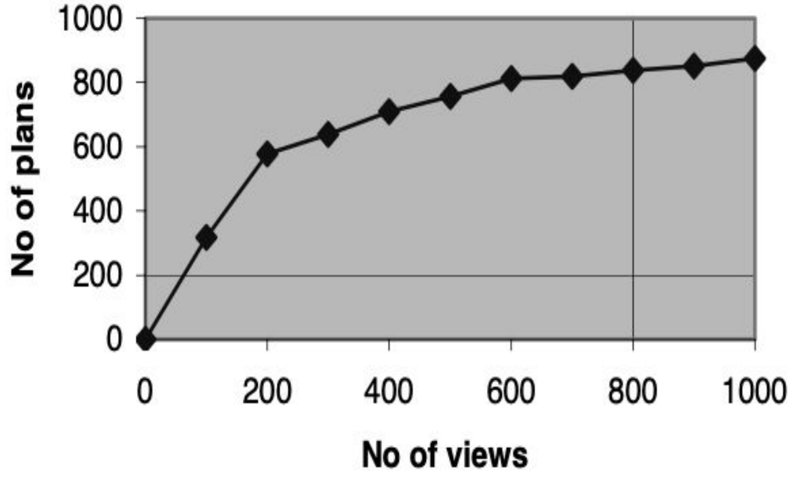


Figure 3: Total increase in optimization time and time spent in view-matching rule



Results - II

Figure 4: No of final query plans using materialized views.


No of plans
No of views