VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Query Rewriting

Lecturer : Dr. Pavle Mogin

SWEN 432
Advanced Database Design and Implementation

Plan for Query Rewriting

- Motives
- Query rewrite methods
  - Full text match,
  - Partial text match,
  - General query rewrite method:
    - Join compatibility
    - Data sufficiency,
    - Grouping compatibility
    - Aggregate computability
- Query rewrite and database constraints
- Query Generator

Motives

- The main task of an OLAP system is to provide fast answers to DSS queries
- Materialized views (MV) are built with that aim
- But user queries are defined in terms of base DW tables
- So, the task of an OLAP query processor is:
  - To check whether a particular query can be answered using some of the existing MVs, and
  - If possible, to transform the query in terms of these views
- We are going to consider rewriting mechanisms of a Relational OLAP (Oracle 9i)

Looking for a View and Query Rewriting

- Given a set of materialized views $\{V_1, \ldots, V_n\}$ and a query Q
- A query processor has to look for one or more views that may be used to evaluate the query, and
  - If at least one such view exists,
  - To rewrite (modify) the query according to the declaration of the view
- Generally, query rewriting is transforming the query in a semantically equivalent form that will produce the same output but may be processed more efficiently

Simple and Complex Materialized Views

- In the ROLAP engine considered, the MVs are classified on:
  - Complex and
  - Simple
- Complex MVs are those that have:
  - Theta joins,
  - `OR` operators in the `WHERE` clause,
  - A `HAVING` clause,
  - Inline views,
  - Multiple instances of the same table or view,
  - Set theoretic operators (`UNION, INTERSECT, MINUS`)
- All other MVs are simple

Example Star Schema

- Tables:
  Shop ({*ShopId*, *ShopName*, *City*, *CityName, Country*}, {*ShopId* })
  Product ({*ProductId*, *ProdName*, *Brand* }, {*ProductId* })
  Time ({*TimeId*, *Date*, *Week*, *Month*}, {*TimeId* })
  Fact ({*ShopId*, *ProductId*, *TimeId*, *Sales*}, {*ShopId* + *ProductId* + *TimeId* })
- Referential integrities:

  Fact [*ShopId*] $\subseteq$ *Shop* [*ShopId*]

  Fact [*ProductId*] $\subseteq$ *Product* [*ProductId*]

  Fact [*TimeId*] $\subseteq$ *Time* [*TimeId*]
- Hierarchies:

  ShopId$\rightarrow$*ShopName*, *ShopId*$\rightarrow$*City*$\rightarrow$*Country*, *City*$\rightarrow$*CityName*
  ProductId $\rightarrow$*ProdName*, *ProductId* $\rightarrow$*Brand*
  TimeId $\rightarrow$*Date*, *TimeId* $\rightarrow$*Month* $\rightarrow$*Quarter*$\rightarrow$*Year*

Two Simple Materialized Views

```
CREATE MATERIALIZED VIEW join_fact_shop_time
AS
SELECT s.ShopId, ShopName, Sales, f.ProductId,
t.TimeId, Date
FROM Fact f NATURAL JOIN Shop s NATURAL JOIN
Time t;
```

```
CREATE MATERIALIZED VIEW sum_fact_shop_prod
AS
SELECT City, ProdName, AVG(Sales) AS avg_sal,
COUNT(Sales) AS count_sal
FROM Fact f NATURAL JOIN Shop s NATURAL JOIN
Product p
GROUP BY City, ProdName;
```

Query Rewrite Methods

- Full text match:
  - Take the text of the query and compare it with the `SELECT` part of a MV
  - If they completely match, rewrite the query in terms of MV
- Partial text match:
  - Start comparing SQL texts of the query and a MV from the FROM clause,
  - If they match, and the view contains sufficient data to compute the answer to the query, rewrite the query in terms of the MV
- Complex MVs are used for query rewrite on the base of text match (full or partial), only

Partial Text Match Rewrite

- Query:

```
SELECT City, ProdName,
SUM(Sales) AS sum_sal FROM
Fact f NATURAL JOIN Product p NATURAL
JOIN Shop
GROUP BY City, ProdName;
```

- Will be rewritten using `sum_fact_shop_prod` MV in the following way:

```
SELECT City, ProdName,
avg_sal * count_sal AS sum_sal
FROM sum_fact_shop_prod;
```

General Query Rewrite Method

- When neither SQL text match succeeds and there is at least one simple MV, the optimizer uses general method
- The general method enables use of a MV even if it contains only part of data requested by the query, or if it contains more data than the query needs
- The general method employs the following four checks:
  - Join Compatibility,
  - Data Sufficiency,
  - Grouping Compatibility, and
  - Aggregate Computability

MV Types versus Rewrite Checks

	MV with Joins Only	MV with Joins and Aggregates	MV with Aggregates on a Single Table
Join Compatibility	✓	✓	
Data Sufficiency	✓	✓	✓
Grouping Compatibility		✓	✓
Aggregate Computability		✓	✓

The Basic Principle of a Query Rewrite

- A query may be rewritten using a MV only if the following is met:
  1. Either all or part of the result requested by a query is obtainable from the precomputed results stored in the MV, and
  2. If only a part of the result is obtainable, then the MV must have data that functionally define missing data and thus these can be retrieved
- The four checks are performed in the given order, and if any fails to satisfy the conditions given above, the MV is abandoned

Join Compatibility Check

- The join compatibility check is satisfied if either of the following is true:
  - The query and a MV join on the same tables using the same join conditions,
  - The MV joins on a subset of the join tables and conditions of the query, and the MV contains columns of the missing join conditions, or
  - The query joins on a subset of the join tables and conditions of a MV
- Otherwise the join compatibility check fails, and there is no reason to proceed checking with that MV

Join Compatibility Example

- The MV `join_fact_shop_time` passes the join compatibility check for the query:

```
SELECT ShopName, ProdName, SUM(Sales)
FROM Fact NATURAL JOIN Shop NATURAL JOIN
Time NATURAL JOIN Product
WHERE Date BETWEEN '14-07-07' AND '31-
08-07'
GROUP BY ShopName, ProdName;
```

The MV join is a subset of query joins (MV does not join the `Product`), but MV contains the `Product` table's primary key

Data Sufficiency Check

- In this check, the optimizer determines if the necessary column data requested by a query can be obtained from a MV
- A MV contains sufficient data for a query if:
  - The set of column data needed by the query is a subset of the set of column data contained in the MV, or
  - Data needed by the query can be inferred from the MV
- For data inference:
  - Data equivalence,
  - Expression matching, and
  - Functional dependencies

  are used

Data Equivalence

- Data equivalence pertains to the equivalence of one column with another one
- For example, if an inner join between tables A and B is based on join condition A.X = B.X, then the data column A.X in a MV can be used to match column B.X in a query

Expression Matching

- A column or an argument of an aggregate can contain an expression like an arithmetic expression:
  - A + B, or B + A, or A – (-B), or…
  - -A*C + A*B, or A*(B - C), or A*B – A*C, or (B - C)*A, or…
- To allow their comparison a query optimizer converts each of them into the same canonical form

Functional Dependencies

- A MV can be used to rewrite a query even if it does not contain data needed by the query, but functionally defines these data
- Information about available functional dependencies is obtainable from the definitions of dimension hierarchies

Data Sufficiency Example

- The MV `join_fact_shop_time` will also pass the data sufficiency check for the query:

```
SELECT ShopName, ProdName, SUM(Sales)
FROM Fact NATURAL JOIN Shop NATURAL
JOIN Time NATURAL JOIN Product
WHERE Date BETWEEN '14-07-07' AND '31-
08-07'
GROUP BY ShopName, ProdName;
```

- Since the MV contains `ShopName, Sales`, and `Date`, while the missing `ProdName` is functionally dependent on `ProdId`, which exists in the MV

Query Rewrite

- So, the query

```
SELECT ShopName, ProdName, SUM(Sales)
FROM Fact NATURAL JOIN Shop NATURAL
JOIN Time NATURAL JOIN Product
WHERE Date BETWEEN '14-07-07' AND '31-08-
07'
GROUP BY ShopName, ProdName;
```

will be rewritten using `join_fact_shop_time` MV as

```
SELECT ShopName, ProdName, SUM(Sales)
FROM join_fact_shop_time NATURAL JOIN
Product
WHERE Date BETWEEN '14-07-07' AND '31-08-
07'
GROUP BY ShopName, ProdName;
```

Another Data Sufficiency Example

- Consider the following query:

```
SELECT City, ProdName, SUM(Sales) AS sum_sal
FROM Fact f NATURAL JOIN Shop s NATURAL
JOIN Product p
WHERE CityName = 'Wellington'
GROUP BY City, ProdName;
```

  and the MV `sum_fact_shop_prod`

- Since the set of the query join tables is equal to the set of MV join tables, the Join Compatibility check is positive
- MV contains *City* and *ProdName* columns, but does not contain *CityName*
- Fortunately, *City*→*CityName* follows from the location dimension hierarchy
- So, Data Sufficiency check is also positive

Another Data Sufficiency Example (cont)

- But, there is a problem, `City` is not a key of the `Shop` table
- Joining back `Shop` with `sum_fact_shop_prod` MV using `City` will be a lossy join
- The query processor will need to apply a join of the following kind:

```
...
FROM sum_fact_shop_prod NATURAL JOIN
(SELECT DISTINCT City, CityName FROM
Shop) AS s
...
```

- The rewrite will occur only if Grouping Compatibility and Aggregate Computability checks prove positive

Grouping Compatibility Check

- This check is required only if both MV and the query contain a GROU BY clause
- Grouping compatibility check relies on the dimension hierarchies
- This check will be positive if for each attribute *A* in the query grouping list there is an attribute *B* in the MV grouping list such that $B \rightarrow A$ follows from the dimension hierarchy definitions
- Note that $B \rightarrow A$ may also be trivial or transitive

Grouping Compatibility Example One

- Suppose a MV grouping list is:

```
...
GROUP BY City, ProdId, Month;
```

- The set of functional dependencies *F* is:

  {*ShopId*→*ShopName*, *ShopId*→*City* →*Country*,
  City→*CityName*
  ProductId →*ProdName*, *ProductId* →*Brand*
  TimeId →*Date*, *TimeId* →*Month* →*Quarter*→*Year* }

- Then, a query list containing any meaningful subset of:

  {*City, CityName, Country*, *ProductId, ProdName*, *Brand*, *Month*, *Quarter*, *Year* }

  will pass the Grouping Compatibility check

Grouping Compatibility Second Example

- Consider query:

```
SELECT Country, ProdName, SUM(Sales) AS sum_sal
FROM Fact f NATURAL JOIN Shop s NATURAL
JOIN Product p
GROUP BY Country, ProdName;
```

  and the MV `sum_fact_shop_prod`
- Join Compatibility check will pass,
- Data Sufficiency check will determine that a join back to the `Shop` table is needed to provide for the attribute `Country` and will pass
- Grouping Compatibility Check will determine that a roll up (from `City` to `Country`) is needed and will pass
- The Aggregate Computability check needs to determine whether it is possible to compute the roll up

Aggregate Computability

- Here the query optimizer determines if the aggregates requested by a query can be derived or computed from aggregates of a MV
- Distributive aggregates are directly computable
  - e.g. SUM, COUNT
- Algebraic aggregates are computable if their fix length p-tuple is available
  - e.g. AVG with (SUM, COUNT), STDEV with (SUM, COUNT), VAR with (SUM, COUNT)
- Holistic aggregates are, by the rule, not computable

Aggregate Computability Example

- Consider the query:

```
SELECT ProdName, AVG(Sales) AS avg_sal
FROM Fact f NATURAL JOIN Product p
GROUP BY ProdName;
```

  and the MV *sum_fact_shop_prod*
- Join Compatibility check will pass,
- Data Sufficiency Check will pass,
- Grouping Compatibility Check will determine that a roll up (from *City, ProdName* to *ProdName*) is needed and will pass
- The Aggregate Computability check will determine that the computation of AVG(Sales) is possible, since AVG(Sales) and COUNT(Sales) are available

Aggregate Computability Example (cont)

- Finally, the query optimizer will rewrite the query as

```
SELECT ProdName,
SUM(avg_sal*count_sal) / SUM(count_sal)  AS avg_sal
FROM sum_fact_shop_prod
GROUP BY ProdName;
```

Query Rewriting and Constraints

Rewrite Check	Functional Dependencies	Primary Key, Foreign key, Not Null
SQL Text Matching		
Join Compatibility		✓
Data Sufficiency	✓	✓
Grouping Compatibility	✓	✓
Aggregate Computability		

A Query or View Generator

- Let *V* be a view and *Q* be a query or a view
- If a query *Q* and a materialized view *V* satisfy:
  - Join Compatibility,
  - Data Sufficiency,
  - Grouping Compatibility, and
  - Aggregate Computability

  then the view *V* can act as a generator of *Q*
- If *V* is a generator of *Q*, then a query optimizer can use *V* to rewrite *Q*
- If there is a set of materialized views *SMV* in a DW then a subset of *SMV* may be the set of generators of a query *Q*

A Query or View Generator Example

- Query Q : Retrieve sales by product_category, city and month
- Set of materialized views SMV :

  {

  View V_1 : sales by product, city and date,

  View V_2 : sales by industry, city and date,

  View V_3 : sales by product, shop and month

  }
- The set of the query Q generators is $\{V_1, V_3\}$
- The view V_1 is a generator for the view V_2

Looking for Query Generators

- When a query *Q* is issued, query optimizer has to look for a materialized view to evaluate the query
- If there is more than one generator for a given query, how to decide on which one to use?

A Question for You

- Which of the materialized views should be chosen to rewrite the query *Q*
  - V_1?
  - V_3?
- Answer:
  - If Average_Number_Of_Shops < Number_Of_Days_In_a_Month, then V_3

Choosing the Best Generator

- The ROLAP Engine, we considered so far, uses the following strategy in deciding which MV to use to rewrite a query:
  - After a MV is chosen for a rewrite, the optimizer performs the rewrite, and then tests whether the rewritten query can be rewritten further with another MV
  - This process continues until no further rewrites are possible
  - The rewritten query is optimized,
  - Original query is optimized, and
  - The optimizer chooses the least costly alternative
  - The cost is estimated using the sum of table cardinalities in each of the solutions

Summary

- One of the tasks of a ROLAP engine is to rewrite queries using materialized views
- Text match methods compare SQL text of a query with definitions of materialized views
- Join Compatibility checks:
  - Whether a query joins are contained in a MV joins, and
  - If not, whether MV can be extended with joins needed
- Data Sufficiency checks:
  - Whether data requested by a query are contained in a MV, and
  - If not, whether MV can be extended with data needed

Summary

- Grouping Compatibility checks:
  - Whether each attribute in a query grouping list is functionally dependent on an attribute in a MV grouping list, and
  - Whether a roll up of aggregates will be needed
- Aggregate Computability checks:
  - Whether the query aggregate data can be inferred from MV aggregate data
- Query rewriting relies on:
  - Functional dependencies given in dimension definitions
  - Primary key, foreign key, and not null constraints
- A query optimizer chooses less costly alternative between the original and a rewritten query