ISIT312 Big Data Management

# Physical Data Warehouse Design

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

# Physical Data Warehouse Design

## Outline

# Techniques for Physical Data Warehouse Design

## Materialized Views

- A view physically stored in the DB
- Typical problems: view update, view selection

## Indexing

- Used in Data Warehouse together with materialized views
- Specific for Data Warehouse: bitmap and join indexes

## Partitioning

- Divides the contents of a relation in several files
- Horizontal and vertical partitioning

# Physical Data Warehouse Design

## Outline

# Materialized Views

Materialized view is a relational table that contains the rows that would be returned by the view definition - usually `SELECT` statement of SQL

If we consider relational views as stored queries then materialized views can be considered as stored results

Materialized views are created and used to reduce an amount of time needed to compute `SELECT` statements, for example join materialized views eliminate the needs to join the relational table

There are two ways how materialized view can be used:

- brute force method
- transparent query rewrite

In brute force method SQL is written to explicitly access the view

Transparent query rewrite method is applied when a query optimizer detects that a query can be computed against a materialized view instead of the source relational tables

# Materialized Views

View maintenance means that when base relational tables are updated, view must be updated too

Incremental view maintenance means that updated view is computed from the individual modifications to the relational tables and not from the entire relational tables

Creating materialized view

Creating materialized view

```
CREATE MATERIALIZED VIEW MV_ORDERS
REFRESH ON COMMIT
ENABLE QUERY REWRITE
AS( SELECT O_ORDERKEY, O_CUSTKEY, O_TOTALPRICE, O_ORDERDATE
    FROM ORDERS
    WHERE O_ORDERDATE > TO_DATE('31-DEC-1986','DD-MON-YYYY') );
```

Direct access to materialized view

Directaccess to materialized view

```
SELECT *
FROM MV_ORDERS
WHERE O_ORDERDATE = TO_DATE('01-JAN-1992','DD-MON-YYYY')
```

# Materialized Views

Access to materialized view through query rewriting

Indirect access to materialized view through query rewriting

```
SELECT O_ORDERKEY, O_CUSTKEY, O_TOTALPRICE, O_ORDERDATE
FROM ORDERS
WHERE O_ORDERDATE > TO_DATE('31-DEC-1986','DD-MON-YYYY');
```

- The results from `EXPLAIN PLAN` statement

Query processing plan

```
PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------
| 0  | SELECT STATEMENT              |           | 108K| 2539K| 507 (1)| 00:00:01 |
|* 1 | MAT_VIEW REWRITE ACCESS FULL| MV_ORDERS | 108K| 2539K| 507 (1)| 00:00:01 |
----------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter("MV_ORDERS"."O_ORDERDATE">TO_DATE(' 1986-12-31 00:00:00',
           'syyyy-mm-dd hh24:mi:ss'))
```

# Physical Data Warehouse Design

## Outline

# Indexes for Data Warehouses

An index provides a quick way to locate data of interest

Sample query

SELECT statement with equality condition in WHERE clause

```
SELECT *
FROM EMPLOYEE
WHERE EmployeeKey = 007;
```

With the help of an index over a column `EmployeeKey` (primary key in `EMPLOYEE` table), a single disk block access will suffice to answer the query

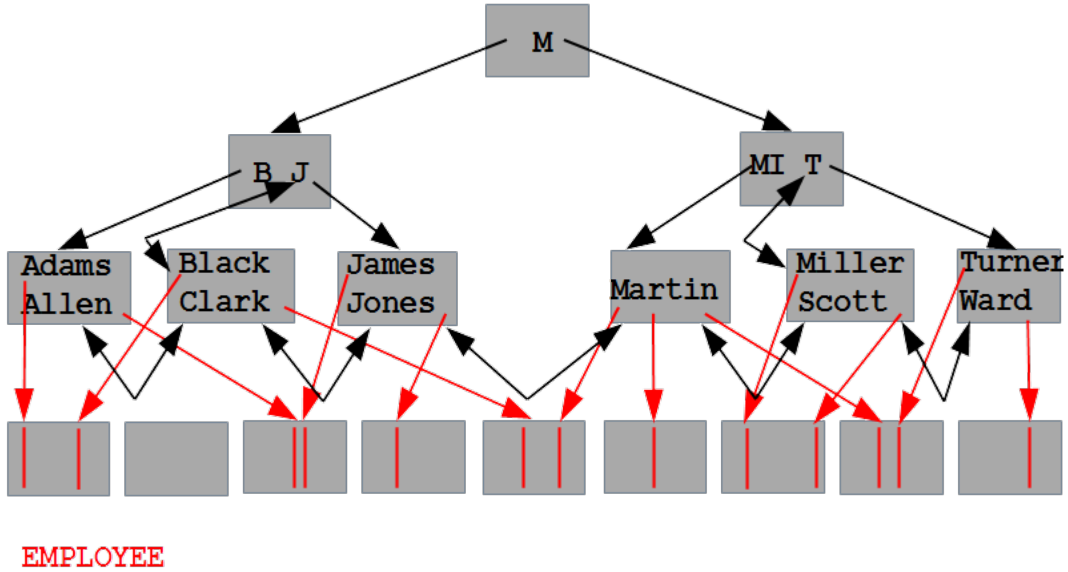
Without this index, we should perform a complete scan of table `EMPLOYEE`

Drawback: Almost every update on an indexed attribute also requires an index update

Too many indexes may degrade performance

Most popular indexing techniques in relational databases include B*-trees and bitmap indexes

# B*-tree index implementation




B*-tree can be traversed either:

- vertically from root to leaf level of a tree
- horizontally either from left corner of leaf level to right corner of leaf level or the opposite
- vertically and later on horizontally either towards left lower corner or right lower corner of leaf level

# Bitmap Indexes

| ProductKey | ProductName | QuantityPerUnit | UnitPrice | Discontinued | CategoryKey |
|---|---|---|---|---|---|
| p1 | prod1 | 25 | 60 | No | c1 |
| p2 | prod2 | 45 | 60 | Yes | c1 |
| p3 | prod3 | 50 | 75 | No | c2 |
| p4 | prod4 | 50 | 100 | Yes | c2 |
| p5 | prod5 | 50 | 120 | No | c3 |
| p6 | prod6 | 70 | 110 | Yes | c4 |

Product dimension table

| | 25 | 45 | 50 | 70 |
|---|---|---|---|---|
| p1 | 1 | 0 | 0 | 0 |
| p2 | 0 | 1 | 0 | 0 |
| p3 | 0 | 0 | 1 | 0 |
| p4 | 0 | 0 | 1 | 0 |
| p5 | 0 | 0 | 1 | 0 |
| p6 | 0 | 0 | 0 | 1 |

Bitmap index for attribute QuantityPerUnit

| | 60 | 75 | 100 | 110 | 120 |
|---|---|---|---|---|---|
| p1 | 1 | 0 | 0 | 0 | 0 |
| p2 | 1 | 0 | 0 | 0 | 0 |
| p3 | 0 | 1 | 0 | 0 | 0 |
| p4 | 0 | 0 | 1 | 0 | 0 |
| p5 | 0 | 0 | 0 | 0 | 1 |
| p6 | 0 | 0 | 0 | 1 | 0 |

Bitmap index for attribute UnitPrice

# Bitmap Indexes: Example

Products having between 45 and 55 pieces per unit, and with a unit price between 100 and 200

| | 45 | 50 | OR1 |
|---|---|---|---|
| p1 | 0 | 0 | 0 |
| p2 | 1 | 0 | 1 |
| p3 | 0 | 1 | 1 |
| p4 | 0 | 1 | 1 |
| p5 | 0 | 1 | 1 |
| p6 | 0 | 0 | 0 |

OR for QuantityPerUnit

| | 100 | 110 | 120 | OR2 |
|---|---|---|---|---|
| p1 | 0 | 0 | 0 | 0 |
| p2 | 0 | 0 | 0 | 0 |
| p3 | 0 | 0 | 0 | 0 |
| p4 | 1 | 0 | 0 | 1 |
| p5 | 0 | 0 | 1 | 1 |
| p6 | 0 | 1 | 0 | 1 |

OR for UnitPrice

| | OR1 | OR2 | AND |
|---|---|---|---|
| p1 | 0 | 0 | 0 |
| p2 | 1 | 0 | 0 |
| p3 | 1 | 0 | 0 |
| p4 | 1 | 1 | 1 |
| p5 | 1 | 1 | 1 |
| p6 | 0 | 1 | 0 |

AND operation

# Indexes for Data Warehouses: Requirements

Symmetric partial match queries

- All dimensions of the cube should be symmetrically indexed, to be searched simultaneously

Indexing at multiple levels of aggregation

- Summary tables must be indexed in the same way as base nonaggregated tables

Efficient batch update

- The refreshing time of a data warehouse must be considered when designing the indexing schema

Sparse data

- Typically, only 20% of the cells in a data cube are nonempty
- The indexing schema must deal efficiently with sparse and nonsparse data

# Physical Data Warehouse Design

## Outline

# Star Queries

Queries over star schemas are called star queries

Join the fact table with the dimension tables

A typical star query: total sales of discontinued products, by customer name and product name

Star query

```
SELECT ProductName, CustomerName, SUM(SalesAmount)
FROM Sales S, Customer C, Product P
WHERE S.CustomerKey = C.CustomerKey AND S.ProductKey = P.ProductKey AND
      P.Discontinued = 'Yes'
GROUP BY C.CustomerName, P.ProductName;
```

Three basic steps to evaluate the query:

- (1) Evaluation of the join conditions
- (2) Evaluation of the selection conditions over the dimensions
- (3) Aggregation of the tuples that passed the filter

# Evaluation of Star Queries with Bitmap Indexes: Example

| Product Key | Product Name | ... | Discontinued | ... |
|---|---|---|---|---|
| p1 | prod1 | ... | No | ... |
| p2 | prod2 | ... | Yes | ... |
| p3 | prod3 | ... | No | ... |
| p4 | prod4 | ... | Yes | ... |
| p5 | prod5 | ... | No | ... |
| p6 | prod6 | ... | Yes | ... |

Product table

| Yes | No |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 0 | 1 |
| 1 | 0 |
| 0 | 1 |
| 1 | 0 |

Bitmap for Discontinued

| Customer Key | Customer Name | Address | Postal Code | ... |
|---|---|---|---|---|
| c1 | cust1 | 35 Main St. | 7373 | ... |
| c2 | cust2 | Av. Roosevelt 50 | 1050 | ... |
| c3 | cust3 | Av. Louise 233 | 1080 | ... |
| c4 | cust4 | Rue Gabrielle | 1180 | ... |

Customer table

| Product Key | Customer Key | Time Key | Sales Amount |
|---|---|---|---|
| p1 | c1 | t1 | 100 |
| p1 | c2 | t1 | 100 |
| p2 | c2 | t2 | 100 |
| p2 | c2 | t3 | 100 |
| p3 | c3 | t3 | 100 |
| p4 | c3 | t4 | 100 |
| p5 | c4 | t5 | 100 |

Sales fact table

| c1 | c2 | c3 | c4 |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

Bitmap for CustomerKey

| p1 | p2 | p3 | p4 | p5 | p6 |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 |

Bitmap for ProductKey

| Yes | No |
|---|---|
| 0 | 1 |
| 0 | 1 |
| 1 | 0 |
| 1 | 0 |
| 0 | 1 |
| 1 | 0 |
| 0 | 1 |

Bitmap join index for Discontinued

# Evaluation of Star Queries using Bitmap Indexes

Evaluation of star query requires

- a B+ tree over `CustomerKey` and `ProductKey`
- Bitmap indexes on the foreign key columns in `Sales` and on `Discontinued` in `Product`

Example of query evaluation

- (1) Obtain the record numbers of the records that satisfy the condition `Discontinued = 'Yes'`
- Answer: Records with `ProductKey` values `p2`, `p4`, and `p6`
- (2) To access the bitmap vectors in `Sales` with these labels perform a join between `Product` and `Sales`
- (3) Vectors labeled `p2` and `p4` match, no fact record for `p6`
- (4) Obtain the values for the `CustomerKey` in these records (`c2` and `c3`)
- (5) Use B+-tree index on `ProductKey` and `CustomerKey` to find the names of products and customers
- (6) Answer: (`cust2,prod2,200`) and (`cust3,prod4,100`)

# Physical Data Warehouse Design

## Outline

# Data Warehouse Partitioning

Partitioning (or fragmentation) divides a table into smaller data sets (each one called a partition)

Applied to tables and indexes

Vendors provide several dfferent partitioning methods

Vertical partitioning splits the attributes of a table into groups that can be independently stored

- E.g., most often used attributes are stored in one partition, less often used attributes in another one
- More records fit into main memory, reducing their processing time

Horizontal partitioning divides a table into smaller tables with same structure than the full table

- For example, if some queries require the most recent data, partition horizontally according to time

# Queries over Partitioned Databases

Partition pruning is the typical way of improving query performance using partitioning

Example: A `Sales` fact table in a warehouse can be partitioned by month

A query requesting orders for a single month only needs to access the partition of such a month

Joins also enhanced by using partitioning:

- When the two tables are partitioned on the join attributes
- When the reference table is partitioned on its primary key
- Large join is broken down into smaller joins

# Partitioning Strategies

Three partitioning strategies: Range partitioning, hash partitioning, and list partitioning

Range partitioning maps records to partitions based on ranges of values of the partitioning key

Time dimension is a natural candidate for range partitioning

Example: A table with a `date` column defined as the partitioning key

- `January-2012` partition will contain rows with key values from January 1 to January 31, 2012

Hash partitioning uses a hashing algorithm over the partitioning key to map records to partitions

- Hashing algorithm distributes rows among partitions in a uniform fashion, yielding, ideally, partitions of the same size
- Typically used when partitions are distributed in several devices, and when data are not partitioned based on time

# Partitioning Strategies

List partitioning specifies a list of values for the partitioning key

Some vendors (e.g. Oracle) support the notion of composite partitioning, combining the basic data distribution methods

Thus, a table can be range partitioned, and each partition can be subdivided using hash partitioning

# References

A. VAISMAN, E. ZIMANYI, Data Warehouse Systems: Design and Implementation, Chapter 7 Physical Data Warehouse Design, Springer Verlag, 2014