Lecture Notes
Week 06

INFS3200 Advanced Database Systems
Semester 1, 2021

Data Warehouse Implementation

Professor Xue Li

The University Of Queensland
AUSTRALIA

+ Last Week

- Data warehouse concept
- Data warehouse design
  - Multidimensional data model
    - Fact and dimensions
    - Star schema, snowflake schema, fact constellation
  - OLAP operations
    - Roll-up, drill-down, slicing, dicing, pivot…
  - Data cube

+ RDBMS vs. Data Warehouse

Aspects	RDBMS	Data Warehouses
Data models	Entity-Relationship	Multidimensional
Schema design	Normalized	De-normalized
Historical data	None or short term	Long term (mths, yrs)
Queries	OLTP	OLAP
Derived data & aggregates	Rare	Common
Data access per operation	A few records	Many records (millions)
Joins	Many	Mostly Pre-joined
Updates	Many small ones	Periodical, bulk, ETL
Workload	Ad hoc, Routine	Batch operations
Indexes	Few	Many

+ Outline

- Data Warehousing
  - Data warehousing concepts
  - Data warehousing design
  - Data warehousing operations
  - Data warehousing implementation
    - Indexing in DW
    - View Materialization in DW
- Data Integration
- Date Quality Management
- Enhanced Data Models

+ Bitmap Indexing

- Index on a particular column
  - Each value in the column has a bit vector
  - The length of the bit vector: # of records in the base table
  - The *i*-th bit is set if the *i*-th row of the base table has the value for the indexed column

What are the advantages of Bitmap Index?

Base table

Cust	Region	Type
C1	Asia	Retail
C2	Europe	Dealer
C3	Asia	Dealer
C4	America	Retail
C5	Europe	Dealer

Index on Region

RecID	Asia	Europe	America
1	1	0	0
2	0	1	0
3	1	0	0
4	0	0	1
5	0	1	0

Index on Type

RecID	Retail	Dealer
1	1	0
2	0	1
3	0	1
4	1	0
5	0	1

+ Example of Bitmap Indexing


```
SELECT          *
FROM    Customer
WHERE Region = 'Europe'  AND
            Type= 'Dealer';
```

Bitwise AND:

Europe: 01001

Dealer: 01101

Result: 01001

Customer

	Cust	Region	Type
0	C1	Asia	Retail
1	C2	Europe	Dealer
0	C3	Asia	Dealer
0	C4	America	Retail
1	C5	Europe	Dealer

Index on Region

RecID	Asia	Europe	America
1	1	0	0
2	0	1	0
3	1	0	0
4	0	0	1
5	0	1	0

Index on Type

RecID	Retail	Dealer
1	1	0
2	0	1
3	0	1
4	1	0
5	0	1

+ Bitmap Indexing

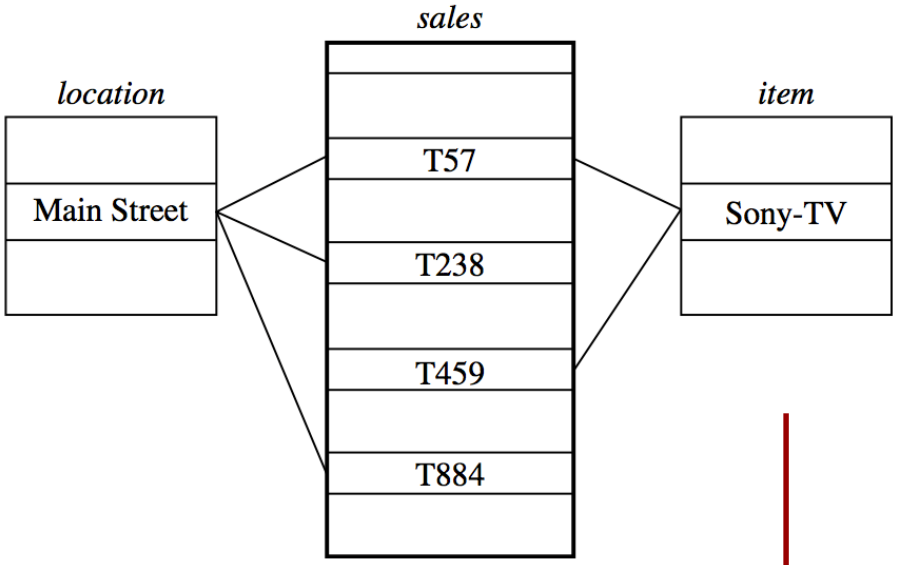
- Advantages
  - Significant reduction in space and I/O
    - E.g., 100 records, varchar(20), 7 unique values: 100*20*8 vs 100*7bits
  - Reduce query processing time
    - Comparison, join and aggregation operations can be reduced to bit operation (How?)
    - Bit operations are very fast
      - “Asia” = “Asia”, “Retail=Retail”? 10 operations
      - Bitmap AND for 8 records 1 operation
- Disadvantages
  - Not suitable for high cardinality domains (Why?)
  - Maintaining a bitmap index takes a lot of resources, therefore, bitmap indexes are only good for the read-only tables or tables that have infrequently updates.

https://www.oracletutorial.com/oracle-index/oracle-bitmap-index/

+ Join Indexing

- Traditional Join index
  - JI(R-id, S-id) where *R(R-id, …)* ⋈ *S(S-id, …)*
  - Map the values to a list of records' IDs
  - Materialize relational join in *JI* file and speed up relational join
- In a data warehouse, join index relates the values of the dimensions of a star schema to rows in the fact table
  - E.g., fact table *Sales* and two dimensions *city* and *product*
    - A join index on *city* maintains for each distinct city a list of R-IDs of the tuples recording the *Sales* in that city
  - Join indices can span multiple dimensions

+ Join Indexing



Join index table for *location/sales*

location	*sales_key*
. . .	. . .
Main Street	T57
Main Street	T238
Main Street	T884
. . .	. . .

Join index table for *item/sales*

item	*sales_key*
. . .	. . .
Sony-TV	T57
Sony-TV	T459
. . .	. . .

Join index table linking *location* and *item* to *sales*

location	*item*	*sales_key*
. . .	. . .	. . .
Main Street	Sony-TV	T57
. . .	. . .	. . .

Join index tables based on the linkages between the *sales* fact table and the *location* and *item* dimension tables

+ Outline

- Data Warehousing
  - Data warehousing concepts
  - Data warehousing design
  - Data warehousing operations
  - Data warehousing implementation
    - Indexing in DW
    - View Materialization in DW
- Data Integration
- Date Quality Management
- Enhanced Data Models

+ View Materialization in DW

- **Advantages**
  - OLAP queries are typically aggregate queries, e.g., CUBE and GROUP BY, Drill-Down, etc.
  - Pre-aggregation is essential for interactive response time
    - Pre-calculate expensive joins
    - Speed up online OLAP queries
- **Disadvantages**
  - It increases storage cost, if dimensionality (d) is big, a large number (2^d) of views would need to be considered for materialization.
  - The content of the materialized views must be maintained when the underlying detail tables are modified.
  - Need carefully designed maintenance strategy to trade-off between query performance and accessibility to up-to-date data.

+ Issues in View Materialization

- What views should we materialize?
- Given a query and a set of materialized views, can we use the materialized views to answer the query?
- How frequently should we refresh materialized views to make them consistent with the underlying tables? (And how can we do this incrementally?)

+ Cuboid

Recall:

- A *fact table T* has *d* dimension attributes A_1,..., A_d, and a numeric value attribute *B*
  - We assume the dimensions have no hierarchies
- A GROUP BY query is parameterized with a subset $G \subseteq \{A_1,..., A_d\}$. Its result, denoted as $\boldsymbol{T_G}$, is referred to as a *cuboid*

A cuboid is a materialized view:

- The *data cubes* of a fact table *T* is the set of all results of the 2^d GROUP BY queries (i.e., there are 2^d cuboids).

+ Running Example

- We will use the following fact table as our running example – **8 possible GROUP BY queries**:

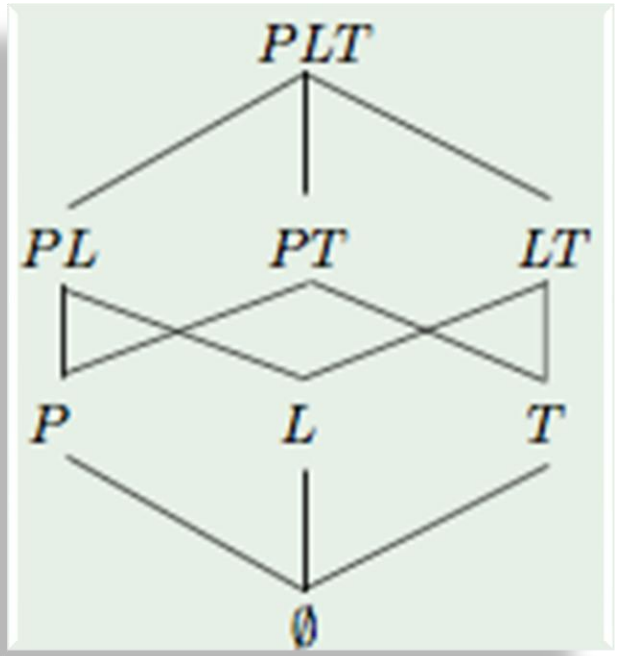
Example 1.

product	location	time	sales
tv	HK	Jan	5
tv	NY	Jan	6
tv	HK	Feb	4
tv	SH	Feb	8
tv	SH	Mar	2
dvd	NY	Jan	3
dvd	SH	Jan	7
dvd	SH	Feb	1
laptop	NY	Feb	4
laptop	SH	Feb	9

+ Lattice

- All the subsets of $\{A_1,...,A_d\}$ form a *lattice L,* where a subset G is a parent of G' if and only if $G' \subset G$ and $|G|=|G'|+1$

sales_cube (2^3)




```
COMPUTE CUBE sales_cube   AS
SELECT                    SUM(*)
FROM                      Sales
CUBE BY poduct, location, time
```

PLT:

SELECT SUM(sales)
FROM Sales
GROUP BY product, location, time

Sales

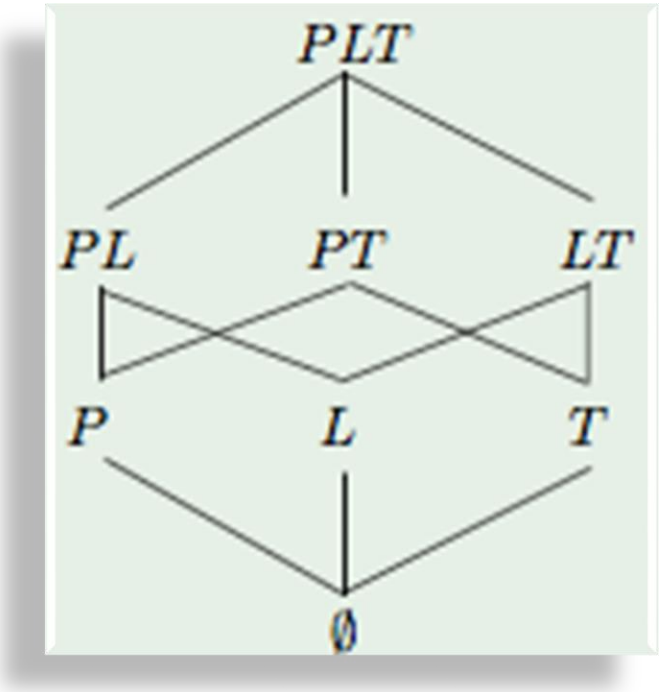
product	location	time	sales

+ Query on Materialized Views (2-1)

- How are queries answered from a set of materialized cuboids? What is the cost?
- Q1: If there are no materialized views, how to answer a GROUP BY query with dimension set G = {product}?
  - Sort **fact table *T*** by *G = {product}*
  - All records in **fact table T** will be scanned.
- Suppose that we have pre-computed the cuboid T_{G1} of G1 = {product, location}. We can now answer a GROUP BY query with dimension set G = {product} directly from T_{G1} , instead of from the **fact table *T***
- Q2: What other GROUP BY queries can benefit from a pre-computed T_{G1}?

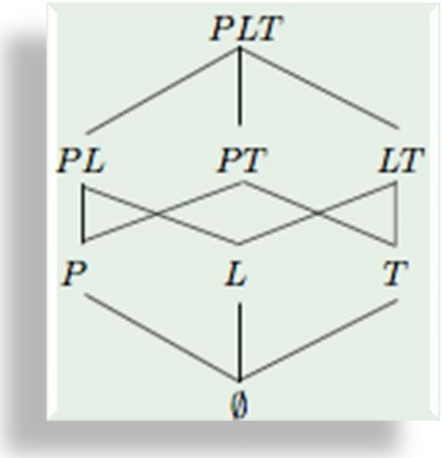
+ Query on Materialized Views (2-2)

- A materialized view with G-cuboid T_G can be used to answer a query with GROUP BY dimension set Q only if $Q \subseteq G$, namely, G is an ancestor of Q in *Lattice L*



+ Some Complications

- Q1: What if the dimensions have hierarchies?
  - **Product**: *item* < *brand*
  - **Location**: *street* < *city* < *state* < *country*
  - **Time**: *day* < *month* < *quarter* < *year*
  - **Query:** *Q* = *{**brand, state**}*
    - Cuboid 1: G_1={year, item, city}?
    - Cuboid 2: G_2={year, brand, country}?
    - Cuboid 3: G_3={year, brand, state}?
    - Cuboid 4: G_4={item, state}?



- Q2: What if some cuboids have indices?

+ Query Cost (2-1)

- Let *cost(Q,G)* be the cost of answering a GROUP BY query with dimension set *Q*, on the materialized view of *G*-cuboid T_G, while $Q \subseteq G$.
- Let us define *cost(Q,G)* as follows:

$$cost(Q,G) = SORT(T_G) \quad \text{if } Q \subseteq G$$

where $SORT(T_G)$ is the cost of sorting the materialized view of cuboid T_G.

Note that:

For any Q_1 and Q_2 , $Q_1 \neq Q_2$

$cost(Q_1,G) = SORT(T_G)$ if $Q_1 \subseteq G$

$cost(Q_2,G) = SORT(T_G)$ if $Q_2 \subseteq G$

+ Query Cost (2-2)

- Now suppose that we have materialized a set *S* of cuboids $\{T_{G1}, \ldots, T_{Gk}\}$
  - Assume that *S* always includes the **fact table *T***, why?
- To answer a GROUP BY query with dimension set *Q*, we should use the cuboid (chosen from the ***Lattice***) as follows:

From all the $T_G \in S$ satisfying $Q \subseteq G$, choose the one from the lattice with the smallest SORT (T_G), denoted as $cost_S(Q, G)$.

- If T_G is the selected cuboid, we define:

$$cost_S(Q, G) = SORT(T_G); \qquad Q \subseteq G .$$

In other words, ***$cost_S(Q, G)$*** is the lowest cost of answering the query *Q* from *S.*

+ Strategies of Materializing Views

- Target of View Materialization
  - Minimize the total cost for all possible GROUP BY queries.
- A naive solution
  - Pre-compute and materialize all possible (i.e., 2^d) cuboids
- What would be the problem?
  - Curse of dimensionality (exponential increment of the cuboids)
    - *d* in practice can be very large
    - Cardinality of a dimension can be large
    - Dimensions may have hierarchies
      - Question: If dimension *i* contains L_i levels, how many cuboids will be created in this case?
- Which cuboids should we choose to materialize?

From Cost to Benefit of Materialized Views

- Assume all GROUP BY queries are issued with the same frequency
- The total cost of answering all possible GROUP BY queries from **set *S* of materialized cuboids** is thus:

$$\begin{cases} \text{allcost}(S) = \sum_{\forall\, Q \subseteq G = \{A_1,\ldots,A_t\}} cost_S(Q, G) \\ \text{cost}(Q, G) = \text{SORT}(T_G) \end{cases}$$

$$\text{allcost}(\{T\}) = 2^t\, \text{SORT}(T)$$

- We will use the following function to measure the quality of *S*:

$$benefit(S) = allcost(\{T\}) - allcost(S)$$

- Recall that any *S* will have to include **fact table *T***. Thus, *benefit(S)* ≥ 0 (why?)

+ Running Example

- The following Lattice shows the sorting cost on each cuboid (e.g, $SORT(T_{\{PLT\}}) = 10{,}000$). Suppose $S = \{T_{\{PLT\}}, T_{\{LT\}}\}$. We give the cost of answering each GROUP BY query as:

$$\text{allcost}(S) = \sum_{\forall\, Q \subseteq G = \{A_{G1},\ldots,A_{Gt}\}} SORT(T_G)$$

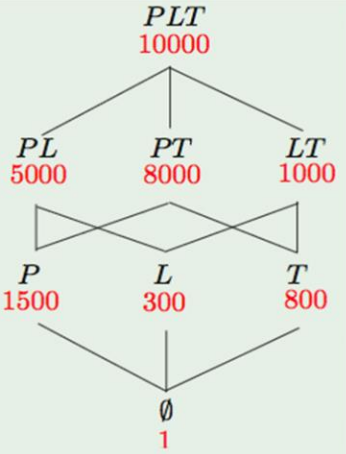
Note that:

For any Q_1 and Q_2 , $Q_1 \neq Q_2$

$cost(Q_1,G)=SORT(T_G)$ if $Q_1 \subseteq G$

$cost(Q_2,G)=SORT(T_G)$ if $Q_2 \subseteq G$

Q	cost
{PLT}	10000
{PL}	10000
{PT}	10000
{LT}	1000
{P}	10000
{L}	1000
{T}	1000
∅	1000



- Therefore, *allcost(S)* = 44,000.
  Since *allcost ({T })* = 80,000, then we have:

benefit(S) = 80,000−44,000=36,000.

+ Running Example (computing)

There are two ways to calculate the benefit:

1. benefit(S) = *(Total_Cost_without_materialization) – (Total_cost_after_materialization)*

   = SORT($T_{\{PLT\}}$)*8 – (SORT($T_{\{PLT\}}$) * 4 + SORT($T_{\{LT\}}$) * 4

   = 80,000 – (40,000+4,000)

   = 36,000

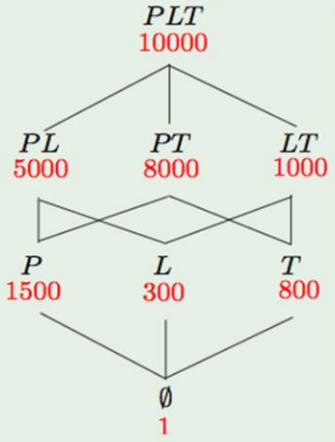
Note that:

For given Q_1 and Q_2 , $Q_1 \neq Q_2$

cost(Q_1,G)=*SORT*(T_G) if $Q_1 \subseteq G$

cost(Q_2,G)=*SORT*(T_G) if $Q_2 \subseteq G$

Q	cost
{PLT}	10000
{PL}	10000
{PT}	10000
{LT}	1000
{P}	10000
{L}	1000
{T}	1000
∅	1000



2. benefit(S) = (Cost_without_materialization_of_a_cuboid) – (Cost_of_its_materialization_saved_with_a_cuboid) * #_of_each_of_its_sub_cuboid

   = (SORT($T_{\{PLT\}}$) – SORT($T_{\{LT\}}$)) * 4

   = (10,000 – 1,000) * 4

   = 36,000

Problem of Choosing Views to Materialize

- View Materialization Problem
  - Given the **fact table T**, the T_G={T_{G1}, T_{G2}, …, T_{Gt}}, an integer *k*, k ≤ t, **find a set *S* of *k* cuboids** (S ⊆ T_G) to be materialized **with the largest *benefit(S).***
- This problem is known to be NP-hard. We will therefore turn to good approximate solutions
  - Greedy algorithm
  - Approximation ratio: *benefit(S)*/*benefit(S*)*=1 − 1/e ≈ 0.632

+ Greedy Algorithm

algorithm Greedy(k)

/* return a set of $k \geq 1$ cuboids */

1. initialize a set S with only one cuboid: the fact table T
2. while $|S| < k$
3. $T_X \leftarrow$ a cuboid T_G maximizing $benefit(S \cup \{T_G\})$ among all the cuboids $T_G \notin S$
4. add T_x to S
5. return S

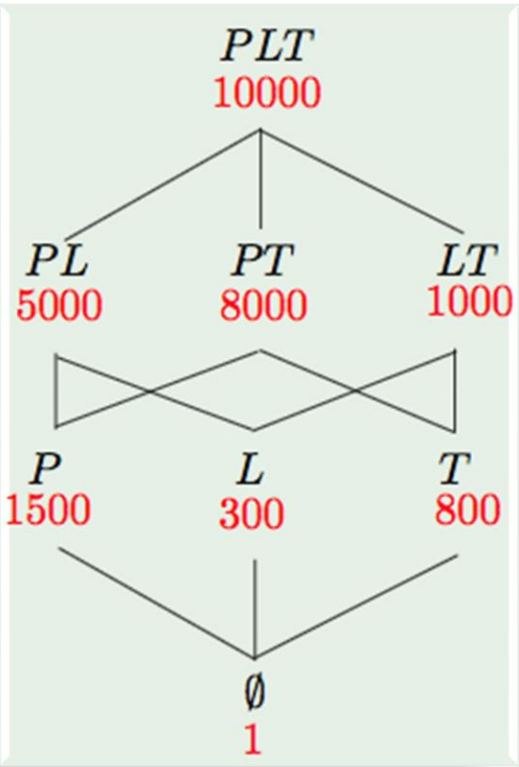
The benefit of selecting a view T_G depends on both the already selected views and the views that can be derived from T_G

+ Running Example

- Let us run the algorithm (by two different ways) with $k = 2$ on the lattice shown on the right. Initially, $S = \{T_{\{PLT\}}\}$

1. benefit (S) = 2^3 SORT(T) – allcost(S U $T_{\{G\}}$)

G	benefit($S \cup \{T_G\}$)	Cost ($S \cup \{T_G\}$)
{PL}	20000	60,000
{PT}	8000	72,000
{LT}	36000	44,000
{P}	17000	63,000
{L}	19400	60,600
{T}	18400	61,600
∅	9999	70,001



2. benefit (S) = (SORT($T_{\{PLT\}}$) – SORT($T_{\{G\}}$)) * # (subsets)

- Therefore, the algorithm adds $T_{\{LT\}}$ to S

- Question: What if queries are issued with different frequency?

+ Conclusions

- When implementing data warehouse, we need to trade-off between storage cost and query efficiency.
- Widely-used indexing techniques in data warehouse systems are: **bitmap index** and **join index.**
- Answer GROUP BY query with materialized views and understand its cost and benefit computations.
- Consider greedy algorithm to choose a set of views to materialize.

Next week: Database Integration

+ Reading Materials

- Jiawei Han, **Data Mining Concepts and Techniques**, 3rd edition (Book available in PDF below)
  - Chapter 4.4: Data Warehouse Implementation
- Michael Teschke et al, **Using Materialized Views To Speed Up Data Warehousing** (download PDF below)
- Kevin Beyer *et al*, **Bottom-Up Computation of Sparse and Iceberg CUBEs** (download PDF below)

http://myweb.sabanciuniv.edu/rdehkharghani/files/2016/02/The-Morgan-Kaufmann-Series-in-Data-Management-Systems-Jiawei-Han-Micheline-Kamber-Jian-Pei-Data-Mining.-Concepts-and-Techniques-3rd-Edition-Morgan-Kaufmann-2011.pdf

https://www.researchgate.net/publication/2323826_Using_Materialized_Views_To_Speed_Up_Data_Warehousing/link/02e7e5232be34a0ba1000000/download

http://www09.sigmod.org/sigmod/sigmod99/eproceedings/papers/beyer.pdf