
OLAP Implementation Issues, Data Warehouses

13 June 2010
Lecture 13

Topics for Today

- Implementation Techniques for OLAP
 - Bitmap Indexes
 - Join Indexes
- Data Warehousing
 - View and Decision Support

Source: RG 25.6-25.8

Aggregation: Percent_Rank

PERCENT_RANK() shows the relative position of a row in the partition:

– $\frac{RANK()-1}{n-1}$ if there are n rows

Meaning: Where we would insert the row into the set if it weren't already there.

Practically, for a n row group:

- Row 1 is 0
- Row n is 1
- Rank increase in equal steps to 1

Not supported in MS SQL Server, only in Oracle

- Could be done using user code and stored procedures

Example: SELECT DeptID, Surname, Salary, Sex, PERCENT_RANK() OVER (PARTITION BY Sex ORDER BY Salary DESC) AS PctRank FROM Employees WHERE State = 'NY';

	DeptID	Surname	Salary	Sex	PctRank
1	200	Martel	55700	M	0.0
2	100	Guevara	42998	M	0.33333333 3
3	100	Soo	39075	M	0.66666666 7
4	400	Ahmed	34992	M	1
5	300	Davidson	57090	F	0
6	400	Blaikie	54900	F	0.33333333 3
7	100	Whitney	45700	F	0.66666666 7
8	400	Wetherby	35745	F	1

Implementation Techniques for OLAP

OLAP applications make interesting variations on the demands for database implementation

We'll talk about some indexing techniques which are appropriate for OLAP

Why are OLAP indexes different?

- OLAP is **mostly read-only**, so indexes won't be updated often
- Indexes must be very fast and able to deal with large quantities of data (10,000 rows, 100,000 rows, etc.)

Bitmap Indexes

A **bitmap index** is a very efficient method for storing **sparse data columns**

- The idea is applicable to many places where sparse data is stored

Sparse data columns are ones which contain data values from a very small set of possibilities

General idea: Prepare an index table with a column for each possible value in the original column c (only a few)

- *IndexTable* (*val1:bit, val2:bit, val3:bit, val4:bit, val5:bit, ...*)

- For a table with n rows, make an n row index table
 - If column c in row m has value X , put a 1 in column X of row m of the index table

Bitmap Index Example

Original Table:

Customers (*custid* integer, *name* varchar(25), *sex* varchar(1), *rating* integer)

- Rating only holds values between 1 and 5 and sex can be either M (male) or F (female)

Original Table

custid	name	sex	rating
112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
113	Woo	M	4

Sex Index

M	F
1	0
1	0
0	1
1	0

Rating Index

1	2	3	4	5
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

Using Bitmap Indexes

Bitmap indexes gives us two advantages over standard B+ indexes

1. You can check AND and OR conditions on columns with just a simple logical AND or OR over the bit values
 - **Example:** Find how many male customers have rating 5.
 - Extract the first column from the sex index and the fifth column from the rating index
 - Perform a bit-wise AND between them
 - **Example:** Find how many customer have rating 2 or 5
 - Extract the second and fifth columns from the rating index
 - Perform a logical OR
2. They are very easily compressed on disk or in memory using standard compression algorithms

When to use them?

Bitmaps are great when there are a few possible values and...

1. We **don't add** too many rows (and care about order)
 2. We **don't delete** too many rows
 - Either we reshuffle the index or mark the row as deleted in the index
 3. There are **many entries** for each possible value
 - If there are only a **few entries** for each value, a normal (value, rid-list) approach is better
-

So for a column/column set with:

1. **Many possible values and many entries per value** → normal tree/hash index
 2. **Few possible values and many read-mostly entries per value** → bitmap index
 3. **Few possible values and few read-mostly entries per value** → (value, rid-list) index
-

Join Indexes

OLAP's goal is to provide fast response time, so it is logical to **precompute common joins (a join index)**

- Saves the time of recomputing each time

Simple idea: For two tables joined on a single attribute, compute pairs $\langle c, p \rangle$

- c is the record id for the first relation
- p is the record id for a corresponding row in the second relation

We can use the file to look up which rids join with which rids

- Classic example of storage space versus speed tradeoff in databases

Join Indexes: Multiple Tables

To generalize to multiple tables, consider the “star schema” of a single fact table and many dimensions

For the fact table F , dimension tables D_1, D_2 with columns C_1, C_2 respectively:

Store a tuple $\langle r_1, r_2, r \rangle$ in the index if the values correspond for a given $r \in F, r_1 \in D_1.C_1, r_2 \in D_2.C_2$

The problem is that we then need to make an index, for each join of interest

Join Indexes: Another Option

Compute the join indexes for each dimension and fact table column

- To do multiple joins on dimensions with the fact table, combine the join indexes for the columns requested

The price: Performance - since we need to combine the join indexes

It's quicker if the join indexes are *bitmap indexes*,
(*bitmapped join indexes*)

- We can compute intersection of the join indexes more simply
- Also means that the columns chosen **must be sparse**

Storing the Relations

Storage becomes an issue when tables are huge and queries must run fast

What can we do?

1. **Vertical partitioning**: If some columns from a table are normally requested in groups
2. **Horizontal partitioning**: To enable regular queries too
3. **MOLAP**: Store the data a multi-dimensional array which is accessed in pieces

So long as it's read only, it doesn't make a difference if we do one or all of the above

So Far

- Implementation Techniques for OLAP
 - Bitmap Indexes
 - Join Indexes
- Data Warehousing
 - View and Decision Support

What is a Data Warehouse?

Data Warehouse: a large (giga, tera, peta) database which dwarfs even the size of commonly used OLTP databases

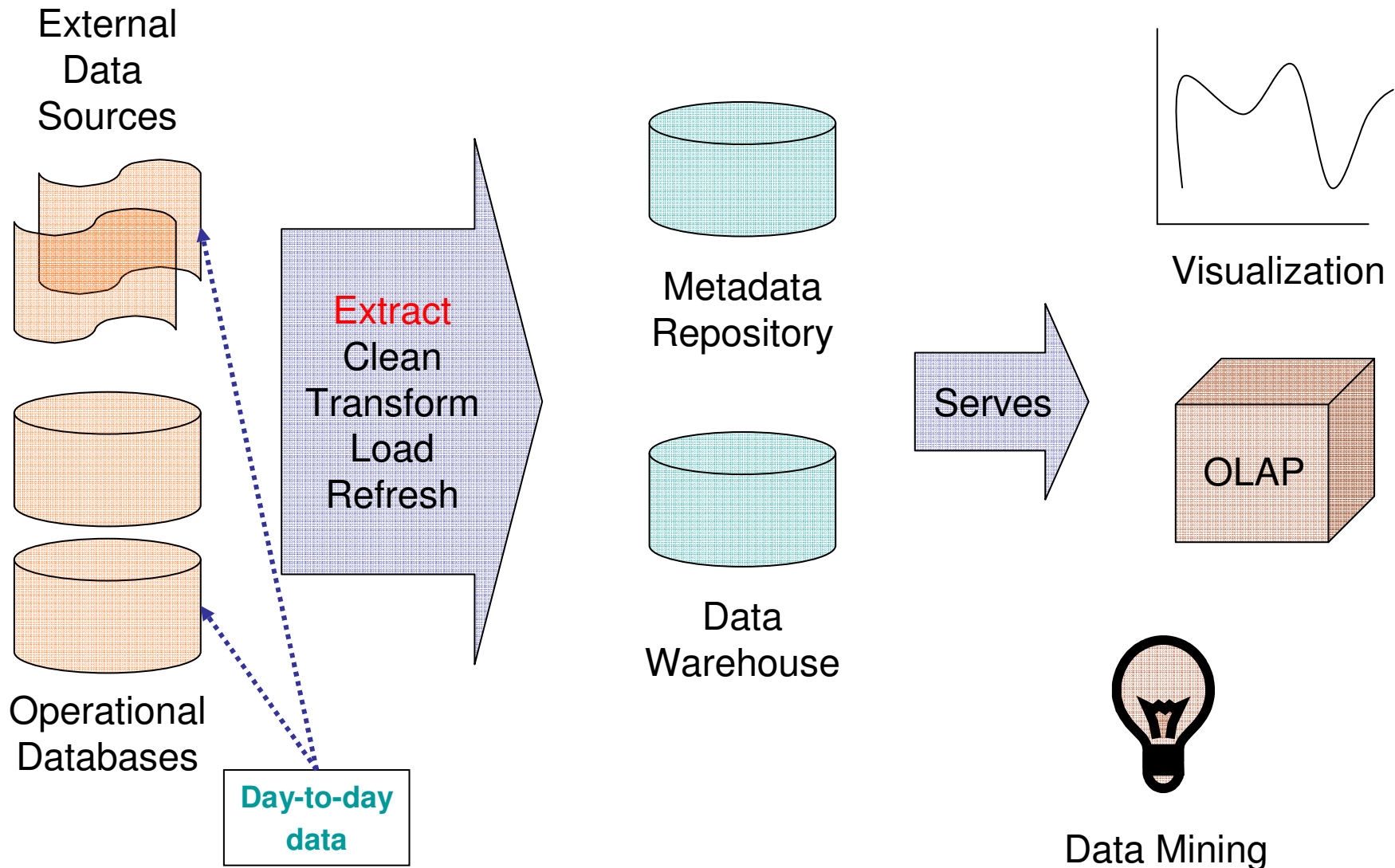
Main Idea: store all of the information in an organization in a single (distributed?) database for analysis and reports

Problems to be solved:

- Efficiency/Speed: Parallelize, Distribute
- Reliability/Uptime
- Update/Maintenance



Creating and Maintaining a Warehouse



Merging is hard, though...

We must unite data from across the enterprise, but this is confounded by some **annoying details**

Unless all of the databases are managed centrally there will be differences between:

- Different keys
- Identifiers
- Currency units
- Time units, time zones
- Normalization levels
- Names of columns, values, and booleans



Missing and partial information is very common as well

Common Problems

- One database stores addresses as **one field** (“123/4 Rehov Dekel”) and another which stores them as **three fields** (“123”, “Rehov Dekel”, “Entrance 4”)
- One stores **Empld**, one stores **Employeeeld**, one stores **Eld**
- One stores an **empty string** for a missing value, one stores **null**

Problems?

Superficial differences can make unifying the databases a pain

- Natural joins won't work

None of these problems must cause the data warehouse to crash, stop working, or fail to import data

Therefore, we need to process data before it's integrated into the warehouse

Importing Data

1. **Extract** the data from the operational and external DBs
2. **Clean** the data
 - Fill in missing values (if needed)
 - Replace illegal or illogical values with something reasonable
3. **Transform** the data to match the data warehouse schema
 - a) Mapping columns from the source tables to the target schema
 - b) Splicing or unifying values
 - c) **Defining views** over the original relations which have been imported
 - i. Removing columns which must be hidden (private data)
 - ii. Removing unneeded extra columns

From now on **use the views instead of the original relations**

- Different from traditional ones since they are **not updated** dynamically as the underlying data changes
- **Take the place** of the original relations

Storing/Loading the Views

The views are loaded into the data warehouse

Loading includes preprocessing too:

- Sorting, indexing, and generation of data statistics and summaries
- Data is not modified once loaded, so doing work up front can make all later uses much faster

Parallelization is essential here because loading large amounts of data into the data warehouse sequentially takes too long

- Many databases include a bulk loader tool to do faster data loads (may ignore constraints or foreign keys)
- Oracle's [SQL*Loader](#) can insert 1 million rows into a table in 41–135 seconds depending on the parameters and error checking options used

Refreshing the Warehouse

The warehouse must be refreshed to ensure that it's up to date

- Recopying relations from the source databases and preparing the new data for inclusion

Old data must be periodically **purged**

- Can mean deletion, but more often it is just backed up to CD, tape, DVD, or other archival media

Refreshing a **distributed warehouse** is more complicated since it will need to be done **asynchronously**

- Since each copy of the source relations are stored in multiple places to speed things up, we have a hard time forcing all changes to all of the copies simultaneously

We'll talk more about maintaining the views (called *materialized views*) in a little bit

Metadata

We need to maintain data about the data in the warehouse

Metadata is stored in the system catalog and describes the relations, views, and summaries that the warehouse stores

- Metadata is normally stored separately from the rest of the data, so it may be considered its own database

The metadata database also has warehouse-specific information such as

- details of data sources
- last refresh date
- administrative information

Views and Decision Support

The value that a data warehouse has is in relation to its speed in answering complex queries

- Want answers in minutes, not days
- The questions it helps answer and the reports that it can generate about its contents

So the tools which we can use to work on the data warehouse will define how helpful it is

- OLAP
- data mining
- information visualization
- reporting services
- Statistical packages

Different Pictures

For efficiency, different user groups of the data warehouse need to see different pictures of the data inside

- We can use views!

Views may filter out **unwanted or excess information**, perform common joins ahead of time, or make summary information available

So we want to make the views run as fast as possible →

View Materialization

- Means some views will live on disk like tables

What to Materialize?

Selecting what to materialize is important since we are dealing with a limited amount of space

- Cost of the space vs. the speedup we will get

Choice based on a cost-benefit analysis

- how often a given view will be used
- how many queries it will speed up
- how much space it takes up on disk

Warehouse Summary

- A data warehouse is characterized by
 - its size
 - its storage of lots of tables
 - its storage of remote data from various sources

A warehouse consists mainly of asynchronously replicated table and periodically synchronized views

Maintaining the replicas and views can be quite some work as we will discuss later

Conclusion

- Implementation Techniques for OLAP
 - Bitmap Indexes
 - Join Indexes
- Data Warehousing
 - View and Decision Support