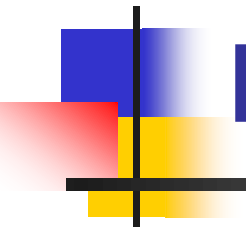
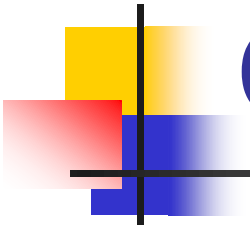


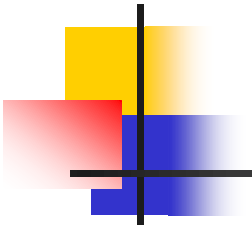
Indexing
L15 (Lec 21)



Outline

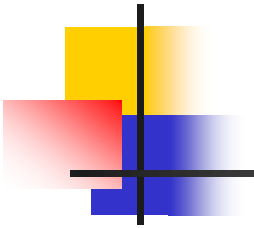
- **Introduction**
- **Why Indexing?**
- **Factors that determine the convenient Indexing technique**
- **Bitmap Indexes**
- **Examples**

Introduction



- The growing interest in Data warehousing for decision-makers is becoming more and more crucial to make faster and efficient decisions
- The problem is that most of the queries in a large data warehouse are complex
- Therefore, many indexing techniques are created to speed up access to data within the tables and to answer ad hoc queries in read-mostly environments.

Introduction



- Indexes are database objects associated with database tables and created to speed up access to data within the table.
- They have already existed in the OLTP relational database system but they cannot handle large amount of data and complex queries that are common in OLAP systems.

Which Indexing technique should be used in a column?

- Factors that determine the convenient

Indexing technique:

1. Cardinality data
2. Distribution
3. Value Range

Characteristics of good indexing technique:

a) The index should be small and utilize space efficiently.

b) The index should be able to operate with other indexes to filter out the records before accessing raw data.

c) The index should support ad hoc and complex queries and speed up join operations.

d) The index should be easy to build (easily dynamically generate), implement and maintain.

Bitmap Indexes

- Bitmap Indexes were first introduced by O'Neil and implemented in the Model 204 DBMS.
- In Bitmap indexes complex logical selection operations can be performed very quickly by applying low-cost Boolean operations such as OR, AND, and NOT on multiple indexes at one time
- thus, reducing search space before going to the primary source data.

- Bitmap indexes are widely used in data warehousing environments
- Bitmap indexing provides:
  - Reduced response time for large classes of ad hoc queries.
  - Reduced storage requirements compared to other indexing techniques.
  - Dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory.
  - Efficient maintenance during parallel DML and loads.

Bitmap Index---Definition

- A collection of bit-vectors of length n
- n means the total number of records
- Each bit of this vector stands for one possible values that may appear in this field.
- At position i for the vector of value v, if the i^{th} record have value v , then the value of vector bit i is "1". otherwise "0"

Indexing OLAP Data: Bitmap Index

- Index on a particular column
- Each value in the column has a bit vector: bit-op is fast
- 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
- not suitable for high cardinality domains

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 1.
How the bitmap index forms

- Sample Relation
  R(F,G)

	F	G
Tuple # 1 →	30	foo
Tuple # 2 →	30	bar
Tuple # 3 →	40	baz
Tuple # 4 →	50	foo
Tuple # 5 →	40	bar
Tuple # 6 →	30	baz

Example 1. How the bitmap index forms (continue..)

- Let's see field "F"
- How many DISTINCT values ?

  3 (30,40,50)

  so, the total number of tuples in the bitmap index is 3.
- How many records ?

  6

  so, the length of bit-vector is 6.

Example 1.
How the bitmap index forms
(continue..)

- Distinct value: 3
- Total tuple : 6
- Bitmap index:

value	Bit-vector
30	xxxxxx
40	xxxxxx
50	xxxxxx

Distinct values:

Length of Bit-Vector is 6

Example 1. How the bitmap index forms (continue..)

- How to form bit-vector ?
- Let's see Value 30
- Which Tuples has the value 30 in field F?

  (1, 2, 6)

value	Bit-vector
30	xxxxxx
40	xxxxxx
50	xxxxxx

Example 1.
How the bitmap index forms
(continue..)

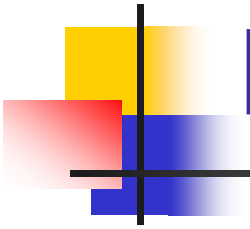
- So, the bit-vector of value 30 is :

Tuple1	Tuple2	Tuple3	Tuple4	Tuple5	Tuple6
1	**1**	**0**	**0**	**0**	**1**

Example 1.
How the bitmap index forms

- Fill the bit-map index of Field "F" as following:

value	Bit-vector
30	1 1 0 0 0 1
40	0 0 1 0 1 0
50	0 0 0 1 0 0



Do we need Bitmap-Index?

- How to find the i^{th} record in a bitmap index?

Advantage of Bitmap Index

- Accelerate the search.
- Example 2
  - Consider following Relation:

Movie (title, year, length, studioName)

  - We need to run the following query:

```
SELECT title
FROM Movie
WHERE studioName = 'Disney' AND
          year = 1995;
```

Example 2. (contiue…)

- With following tuples:

studioName	*year*
Disney	1995
MGM	1996
DreamFactory	2000
MGM	1995
DreamFactory	1996
Disney	2000

Example 2. (contiue…)

- Suppose we build Bitmap index in the field of "studioName" and "year"

Bitmap Index on Field "studioName"

Disney	1 0 0 0 0 1
MGM	0 1 0 1 0 0
DreamFactory	0 0 1 0 1 0

Bitmap Index on Field "studioName"

1995	1 0 0 1 0 0
1996	0 1 0 0 0 0
2000	0 0 1 0 0 1

Example 2. (contiue…)

- The query is :

```
SELECT title
FROM Movie
WHERE studioName = 'Disney' AND
                   year = 1995;
```

- So we INTERSECT the bitmap index with value of 'Disney" and '1995'

```
        1 0 0 0 0 1
AND     1 0 0 1 0 0
        -----------
        1 0 0 0 0 0
```

- Answer: tuple #1

Example 3

- Bitmap index also accelerate range query.
- Consider following Relation R(age, salary)

Age	***salary***
25	60
22	55
30	70
22	55
23	55
25	100
23	45
30	45

Example 3 (continue…)

- We have following bitmap index:

Field “AGE”

22	0 1 0 1 0 0 0 0
23	0 0 0 0 1 0 1 0
25	1 0 0 0 0 1 0 0
30	0 0 1 0 0 0 0 1

Field “salary”

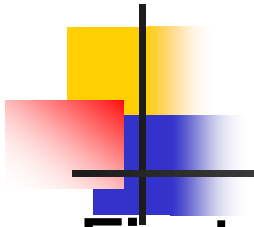
45	0 0 0 0 0 0 1 1
60	1 0 0 0 0 0 0 0
55	0 1 0 1 1 0 0 0
70	0 0 1 0 0 0 0 0
100	0 0 0 0 0 1 0 0

Example 3 (continue…)

- Consider the following Query:

```
SELECT *
FROM R
WHERE   23 <= age <= 25
 and    50 <= salary <= 70
```

Example 3 (continue…)



First , find bitmap index of field "AGE" satisfy the requirement, do "OR" Operation:

```
23          0 0 0 0 1 0 1 0
25  OR      1 0 0 0 0 1 0 0
            1 0 0 0 1 1 1 0
```

Second, find bitmap index of field "Salary" satisfy the requirement, do "OR" operation

```
60          1 0 0 0 0 0 0 0
55          0 1 0 1 1 0 0 0
70  OR      0 0 1 0 0 0 0 0
            1 1 1 1 1 0 0 0
```

Example 3 (continue…)

- Then , do INTERSECT of the 2 vector:

```
    1 0 0 0 1 1 1 0
AND 1 1 1 1 1 0 0 0
    ---------------
    1 0 0 0 1 0 0 0
```

- So, the answer to this query is tuple #1 and tuple #5

Bitmap Index – Summary

- With efficient hardware support for bitmap operations (AND, OR, XOR, NOT), bitmap index offers better access methods for certain queries
  - e.g., selection on two attributes
- Some commercial products have implemented bitmap index
- Works poorly for high cardinality domains since the number of bitmaps increases
- Difficult to maintain - need reorganization when relation sizes change (new bitmaps)

Bitmap Indexes

- Bitmap indexes are widely used in data warehousing environments which typically have large amounts of data and ad hoc queries
- For such applications, bitmap indexing provides:
  - Reduced response time for large classes of ad hoc queries
  - Reduced storage requirements compared to other indexing techniques
  - Dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory
- Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of space because the indexes can be several times larger than the data in the table
- Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

Bitmap Join Indexes

- In addition to a bitmap index on a single table, you can create a bitmap join index (BJI), which is a bitmap index for the join of two or more tables.
- A BJI is a space efficient way of reducing the volume of data that must be joined by performing restrictions in advance.
- For each value in a column of a table, a bitmap join index stores the rowids of corresponding rows in one or more other tables.

Bitmap Join Indexes

- In a data warehousing environment, the join condition is an equi-inner join between the primary key column or columns of the dimension tables and the foreign key column or columns in the fact table.
- Bitmap join indexes are much more efficient in storage than materialized join views, an alternative for materializing joins in advance.
- This is because the materialized join views do not compress the rowids of the fact tables.

Dimensions

- A dimension is a structure that categorizes data in order to enable users to answer business questions.
- Commonly used dimensions are customers, products, and time.
- In Oracle Database, the dimensional information itself is stored in a dimension table.
- In addition, the database object dimension helps to organize and group dimensional information into hierarchies.
- This represents natural 1:n relationships between columns or column groups (the levels of a hierarchy) that cannot be represented with constraint conditions

Dimensions

- Dimensions do not have to be defined. However, if your application uses dimensional modeling, it is worth spending time creating them as it can yield significant benefits, because they help query rewrite perform more complex types of rewrites
- In spite of the benefits of dimensions, you must not create dimensions in any schema that does not fully satisfy the dimensional relationships

Dimensions

- Before you can create a dimension object, the dimension tables must exist in the database possibly containing the dimension data
- You create a dimension using the CREATE DIMENSION statement
- For example, you can declare a dimension products_dim, which contains levels product, subcategory, and category:

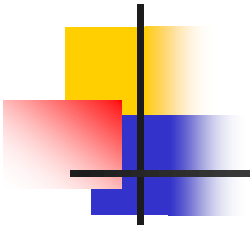
```
CREATE DIMENSION products_dim
LEVEL product IS (products.prod_id)
LEVEL subcategory IS (products.prod_subcategory)
LEVEL category IS (products.prod_category) ...
```

Dimensions

CREATE DIMENSION products_dim
LEVEL product IS (products.prod_id)
LEVEL subcategory IS (products.prod_subcategory)
LEVEL category IS (products.prod_category) ...

- Next step is to specify the hierarchy:

HIERARCHY prod_rollup
(product CHILD OF
subcategory CHILD OF
category)

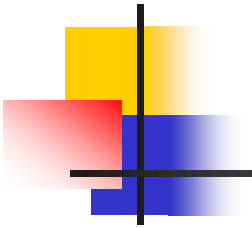


Window Queries in

- **Time dimension is very important in decision support**
- **Queries involving trend analysis have been difficult to express in SQL**
- **A fundamental extension called a "query window" is introduced**

Window Queries

- The WINDOW clause in SQL allows us to write such queries over a table viewed as a sequence (implicitly, based on user-specified sort keys)
- Also referred to as "querying sequences"
- WINDOW clause intuitively identifies an ordered 'window' of rows 'around' each tuple in a table
- We can apply a rich collection of aggregate functions to the window of a row and extend the row with the results
- For example, we can associate the avg. sales over the past 3 days with every sales tuple (daily granularity)
- This gives a 3-day moving avg. of sales



WINDOW & GROUP BY

- Like the WINDOW operator, GROUP BY allows us to create partitions of rows and apply aggregate function such as SUM to rows in a partition
- Unlike WINDOW, there is a single output row for each partition, rather than one output row for each row, and each partition is an unordered collection of rows
- COMPARE with CUBE!!!!

WINDOW: Example

```
SELECT L.state, T.month, AVG(S.sales) OVER W AS movavg
FROM  Sales S, Times T, Locations L
WHERE S.timeid=T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.state
             ORDER BY T.month
             RANGE BETWEEN INTERVAL `1' MONTH PRECEDING
             AND INTERVAL `1' MONTH FOLLOWING)
```

- FROM & WHERE clauses proceed as usual to generate an intermediate table, TEMP.
- WINDOWS are created over TEMP
- 3 steps in defining a window
  - Define partitions of the table (Partitions are similar to groups created by GROUP BY)
  - Specify the ordering of rows within a partition
  - Frame WINDOW: establish the boundaries of the window associated with each row in terms of ordering of rows within partitions

WINDOW: Example

```
SELECT L.state, T.month, AVG(S.sales) OVER W AS movavg
FROM  Sales S, Times T, Locations L
WHERE S.timeid=T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.state
        ORDER BY T.month
        RANGE BETWEEN INTERVAL `1' MONTH PRECEDING
        AND INTERVAL `1' MONTH FOLLOWING)
```

- **Define partitions of the table (Partitions are similar to groups created by GROUP BY)**
- **Specify the ordering of rows within a partition**
- **Frame WINDOW: establish the boundaries of the window associated with each row in terms of ordering of rows within partitions**
- **Window for each row includes the row itself, plus all rows whose month values are within a month before or after.**
- **A row whose month value is June 2006 has a window containing all rows with month = May, June, or July 2006**

WINDOW: Example

```
SELECT L.state, T.month, AVG(S.sales) OVER W AS movavg
FROM  Sales S, Times T, Locations L
WHERE S.timeid=T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.state
             ORDER BY T.month
             RANGE BETWEEN INTERVAL `1' MONTH PRECEDING
             AND INTERVAL `1' MONTH FOLLOWING)
```

- **Answer rows to each row is constructed first by identifying its WINDOW**
- **Then, for each answer column defined using a window agg. fn, we compute the agg. Using the rows in the WINDOW**
- **Each row of TEMP is a row of sales, tagged with extra details about time & location dimensions**
- **One partition for each state and every row of temp belongs to exactly one partition.**

Top N Queries

- If you want to find the 10 (or so) cheapest cars, it would be nice if the DB could avoid computing the costs of all cars before sorting to determine the 10 cheapest.
  - Idea: Guess at a cost c such that the 10 cheapest all cost less than c, and that not too many more cost less. Then add the selection cost<c and evaluate the query.
    - If the guess is right, great, we avoid computation for cars that cost more than c.
    - If the guess is wrong, need to reset the selection and recompute the original query.

Top N Queries

```
SELECT P.pid, P.pname, S.sales
FROM Sales S, Products P
WHERE S.pid=P.pid AND S.locid=1 AND S.timeid=3
ORDER BY S.sales DESC
OPTIMIZE FOR 10 ROWS
```

- **OPTIMIZE FOR construct is not in SQL:92 & not even in SQL:1999!**
- **Supported by IBM's DB2 & Otacle 9i has similar constructs**
- **Compute sales only for those products that are likely to be in TOP 10**

Top N Queries

```
SELECT P.pid, P.pname, S.sales
FROM Sales S, Products P
WHERE S.pid=P.pid AND S.locid=1 AND S.timeid=3
        AND S.sales > c
ORDER BY S.sales DESC
```

- **Cut-off value c is chosen by optimizer using the histogram on the sales column of the sales relation**
- **Much faster approach**
- **Issues:**
  - **How to choose c?**
  - **What if we get more than 10 products?**
  - **What if we get less than 10 products?**

Thank You