Physical Design
Physical Design

• Workload Characterization:
  – Queries with aggregates that require access to a large portion of each table
  – Read-only access
  – Periodic update of data with eventual rebuilding of access physical structures (indexes, views, etc.)

• Physical Structures
  – Non-traditional types of indexes
    • Bitmap indexes, Join indexes, Bitmapped join indexes
    • $B^+$-tree index is not well-suited for:
      – Attributes with low cardinality domain
      – Queries with low selectivity
  – Materialized views:
    • Require the presence of an optimizer able to exploit them
Physical Design

• Optimizer characteristics
  – Must consider statistics while defining data access strategy (cost based)
  – Capability of aggregate navigation

• Physical design procedure
  – Selection of suited data structures to support the most frequent queries (or the most relevant)
  – Choice of structures that contribute to improve more queries at a time
  – Constraints:
    • Disk space
    • Available time for updating data
Physical Design

• Tuning:
  – *A posteriori* variation of support physical structures
  – Requires tools for workload monitoring
  – Often required for OLAP applications

• Parallelism
  – Data fragmentation
  – Queries parallelization
    • inter-query
    • intra-query
  – Join and Group By operations suitable to parallel execution
Physical Access Structures

• Physical access structures describe how data is stored on disk to provide efficient query execution
  – SQL select, update, …

• In relational systems
  – Physical data storage
    • Sequential structures (heap file, ordered sequential structure)
    • Hash structures
  – Indexing to increase access efficiency
    • Tree structures (B-Tree, B⁺-Tree)
    • Unclustered hash index
    • Bitmap index
Heap file

- Tuples are sequenced in *insertion order*
  - insert is typically an *append* at the end of the file
- *All* the space in a block is completely exploited before starting a new block
- Delete or update may cause wasted space
  - Tuple deletion may leave unused space
  - Updated tuple may not fit if new values have larger size
- Sequential reading/writing is very efficient
- Frequently used in relational DBMS
  - jointly with unclustered (secondary) indices to support search and sort operations
Ordered sequential structures

- The order in which tuples are written depends on the value of a given key, called **sort key**
  - A sort key may contain one or more attributes
    - the sort key may be the primary key

- Appropriate for
  - Sort and group by operations on the sort key
  - Search operations on the sort key
  - Join operations on the sort key
    - when sorting is used for join
Ordered sequential structures

• Problem
  – preserving the sort order when inserting new tuples
    • it may also hold for update
• Solution
  – Leaving a percentage of free space in each block during table creation
    • On insertion, dynamic (re)sorting in main memory of tuples into a block
• Alternative solution
  – Overflow file containing tuples which do not fit into the correct block
Ordered sequential structures

• Typically used with B+-Tree clustered (primary) indices
  – the index key is the sort key
• Used by the DBMS to store intermediate operation results
B+Tree

- Provide “direct” access to data based on the value of a key field
  - The key includes one or more attributes
- B stands for *balanced*
  - Leaves are all at the same distance from the root
  - Access time is constant, regardless of the searched value
- Unclustered
  - The leaf contains physical pointers to actual data
    - The position of tuples in a file is totally unconstrained
- Clustered
  - The tuple is contained into the leaf node
    - Constrains the physical position of tuples in a given leaf node
    - Typically used for primary key indexing
Example: Unclustered B⁺-Tree index

STUDENT (StudentId, Name, Grade)

DATA FILE FOR STUDENT TABLE
Example: Clustered B⁺-Tree index

STUDENT (StudentId, Name, Grade)

DATA FILE FOR STUDENT TABLE
Advantages and disadvantages

• Advantages
  – Very efficient for range queries
  – Appropriate for sequential scan in the order of the key field
    • Always for clustered, not guaranteed otherwise

• Disadvantages
  – Insertions may require a split of a leaf
    • possibly, also of intermediate nodes
    • computationally intensive
  – Deletions may require merging uncrowded nodes and re-balancing
Bitmap Index

• Composed of a bit matrix
  – A column for each different value of the indexed attribute domain
  – A row for each tuple (RID in the table)
  – The position \((i,j)\) is 1 if the tuple \(i\) has value \(j\), 0 otherwise

Example: Index on the field Position in the Employee table
Engineer – Consultant – Manager – Programmer – Assistant – Accountant

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Taken from Golfarelli, Rizzi,”Data warehouse, teoria e pratica della progettazione”, McGraw Hill 2006
Bitmap index

• It guarantees direct and efficient access to data based on the value of a **key field**
  – It is based on a **bit matrix**

• The bit matrix references data rows by means of RIDs (Row IDentifiers)
  – Actual data is stored in a separate structure
  – Position of tuples is not constrained
Bitmap index

• The bit matrix has
  – One column for each different value of the indexed attribute
  – One row for each tuple
• Position (i, j) of the matrix is
  – 1 if tuple i takes value j
  – 0 otherwise

<table>
<thead>
<tr>
<th>RID</th>
<th>Val₁</th>
<th>Val₂</th>
<th>...</th>
<th>Valₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: Bitmap index

**EMPLOYEE** (EmployeeId, Name, Job)

Domain of Job attribute = \{Engineer, Consultant, Manager, Programmer, Secretary, Accountant\}

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: Index on the field Position in the Employee table

Engineer – Consultant – Manager – Programmer – Assistant – Accountant
Bitmap index

• Advantages
  – Very efficient for boolean expressions of predicates
    • Reduced to bit operations on bitmaps
  – Appropriate for attributes with limited domain cardinality

• Disadvantages
  – Not used for continuous attributes
  – Required space grows significantly with domain cardinality
**Bitmap Index**

- Well-suited for dimensional attributes with low-cardinality domain
  - Storage requires limited space
  - If domain cardinality (NK) grows, the required space grows as well

<table>
<thead>
<tr>
<th>B-tree</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>NR×<em>Len</em>(Pointer)</td>
<td>NR × NK × 1 bit</td>
</tr>
</tbody>
</table>

*Len*(Pointer) = 4×8 bit

Taken from Golfarelli, Rizzi,”Data warehouse, teoria e pratica della progettazione”, McGraw Hill 2006
**Bitmap Index**

- Efficient for verifying Boolean expressions of predicates
  - Bit-wise and/or on bitmaps

**Example:** “*How many males in Romagna are insured?*”

<table>
<thead>
<tr>
<th>RID</th>
<th>Gender</th>
<th>Ins.</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
<td>No</td>
<td>LO</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>Yes</td>
<td>E/R</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>No</td>
<td>LA</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>Yes</td>
<td>E/R</td>
</tr>
</tbody>
</table>

![Bitmap Index Example](image)

Join index

- Precomputes the join between two tables
  - Stores the RID couples of tuples that satisfy the join predicate

<table>
<thead>
<tr>
<th>Table</th>
<th>Data</th>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sale</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RID couples that satisfy Join condition:
Sells. IDStore = Store.IDStore

**Star index**

- Precomputes the join between two or more tables
  - Stores the RID n-uples of the tuples that satisfy the join predicate

Star index

• Advantages
  – Efficient computation of Joins involving initial index columns (or all columns)

• Disadvantages
  – Useful only for specific Join combinations
    • It is necessary to store a high number of indexes in order to achieve generalization
  – The storage space may become big
    • Joins always include the fact table
Bitmapped join index

- Bit matrix that precomputes the join between a dimension and the fact table
  - A column for each dimension RID
  - A row for each fact table RID
  - The position \((i,j)\) is 1 if the tuple \(i\) of the dimension is joined with the tuple \(j\) of the fact table, 0 otherwise

- Can be used together with traditional bitmap indexes to compute complex queries with conditions on dimensions and multiple joins

```
| RID | 1   | 2   | 3   | ...
|-----|-----|-----|-----|-----
| 1   | 1   | 0   | 0   | ... |
| 2   | 0   | 1   | 0   | ... |
| 3   | 0   | 0   | 1   | ... |
| 4   | 0   | 1   | X   | ... |
| 5   | 1   | 0   | 0   | ... |
|     | ... | ... | ... |     |
```

The row 4 of table SELLS is joined with the row 2 of the table STORE

Taken from Golfarelli, Rizzi,"Data warehouse, teoria e pratica della progettazione", McGraw Hill 2006
Bitmapped join index

Executing a bit-wise OR, the system obtains RID$_i$ that satisfy all conditions for a dimensional table.

Bitmapped join index

$$FT.a_i = DT_i.a_i$$

Bitmap Index on attribute DT$_i$.b$_i$

<table>
<thead>
<tr>
<th>RID</th>
<th>Val$_1$</th>
<th>Val$_2$</th>
<th>...</th>
<th>Val$_t$</th>
<th>...</th>
<th>Val$_h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

For each dimensional table, the system identifies RIDs that satisfy eventual conditions on dimensional attributes.

For each bitmapped join index, the system loads the bit vectors corresponding to the selected RIDs.

Executing a bit-wise OR, the system obtains RID$_i$ that satisfy all conditions for a dimensional table.

$$RID_i = RID_4 \lor RID_5$$

Taken from Golfarelli, Rizzi, “Data warehouse, teoria e pratica della progettazione”, McGraw Hill 2006
### Bitmapped join index

**RID**

<table>
<thead>
<tr>
<th><strong>RID</strong></th>
<th><strong>RID</strong></th>
<th><strong>RID</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**RIDs that satisfy all conditions**

The fact table tuples that satisfy the query are computed with a bit-wise AND between the $n$ vector previously created.

Taken from Golfarelli, Rizzi,”Data warehouse, teoria e pratica della progettazione”, McGraw Hill 2006
Hash structure

• It guarantees direct and efficient access to data based on the value of a *key field*
  – The hash key may include one or more attributes

• Suppose the hash structure has B blocks
  – The hash function is applied to the key field value of a record
    • It returns a value between 0 and B-1 which defines the position of the record
  – Blocks should never be completely filled
    • To allow new data insertion
Example: hash index

STUDENT (StudentId, Name, Grade)

<table>
<thead>
<tr>
<th>Block 0</th>
<th>T1 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK 1</td>
<td>T4 75</td>
</tr>
<tr>
<td>BLOCK 2</td>
<td></td>
</tr>
</tbody>
</table>

DATA FILE FOR STUDENT TABLE

Example:

TUPLE T1
StudentId = 50
H(StudentId = 50) = 1

TUPLE T4
StudentId = 75
H(StudentId = 75) = 1
Hash index

- Advantages
  - Very efficient for queries with equality predicate on the key
  - No sorting of disk blocks is required

- Disadvantages
  - Inefficient for range queries
  - Collisions may occur
Unclustered hash index

• It guarantees direct and efficient access to data based on the value of a key field
  – Similar to hash index

• Blocks contain pointers to data
  – Actual data is stored in a separate structure
  – Position of tuples is not constrained to a block
  • Different from hash index
Example: Unclustered hash index

STUDENT (StudentId, Name, Grade)

TUPLE T1
GRADE = 30
H(GRADE=30)=1

TUPLE T2
GRADE = 40
H(GRADE=40)=1

INDEX BLOCKS

30 → T1
40 → T2

DATA FILE FOR STUDENT TABLE
Index choice

• Indexing of dimensions
  – Attribute frequently involved in selection predicates
  – If the domain has high cardinality, B-tree index
  – If the domain has low cardinality, bitmap index

• Indexes for Join
  – It is seldom necessary to index only external keys of the fact table
  – Be careful when using Star Join Indexes (problems related to column ordering)
  – Bitmapped join index are recommended

• Indexes for Group By
  – Use Materialized Views