# Processing of Massive Datasets

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Bachelor studies, eighth semester
Academic year 2018/19 (summer semester)

## Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.

## Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Summarization, materialization, and denormalization.

# Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Summarization, materialization, and denormalization.
- Data access:

# Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Summarization, materialization, and denormalization.
- Data access: hashing and sorting ($\rightarrow$ tree-based indexing).

# Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Summarization, materialization, and denormalization.
- Data access: hashing and sorting ($\rightarrow$ tree-based indexing).
- Advanced data structures: multi-dimensional indexes, inverted lists, bitmaps, special-purpose indexes.

## Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Summarization, materialization, and denormalization.
- Data access: hashing and sorting ($\rightarrow$ tree-based indexing).
- Advanced data structures: multi-dimensional indexes, inverted lists, bitmaps, special-purpose indexes.
- Data compression.

## Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Summarization, materialization, and denormalization.
- Data access: hashing and sorting ($\rightarrow$ tree-based indexing).
- Advanced data structures: multi-dimensional indexes, inverted lists, bitmaps, special-purpose indexes.
- Data compression.
- Approximate query processing.

# Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Summarization, materialization, and denormalization.
- Data access: hashing and sorting ($\rightarrow$ tree-based indexing).
- Advanced data structures: multi-dimensional indexes, inverted lists, bitmaps, special-purpose indexes.
- Data compression.
- Approximate query processing.
- Probabilistic data structures and algorithms.

# Processing of massive data sets

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Summarization, materialization, and denormalization.
- Data access: hashing and sorting ($\rightarrow$ tree-based indexing).
- Advanced data structures: multi-dimensional indexes, inverted lists, bitmaps, special-purpose indexes.
- Data compression.
- Approximate query processing.
- Probabilistic data structures and algorithms.
- Partitioning and sharding (Map-Reduce, distributed databases).

# Outline

1. Physical storage and data access
2. Materialization
3. Summary

# Outline

1 Physical storage and data access

2 Materialization

3 Summary

# Physical storage

- How to store the data below:

| Year | Products | Sales |
|---|---|---|
| 2010 | Mountain | 5076 |
| 2010 | Road | 4005 |
| 2010 | Touring | 3560 |
| 2011 | Mountain | 6503 |
| 2011 | Road | 4503 |
| 2011 | Touring | 3445 |

## Physical storage

- How to store the data below:

| Sales | Products | | |
|---|---|---|---|
| Year | Mountain | Road | Touring |
| 2010 | 5076 | 4005 | 3560 |
| 2011 | 6503 | 4503 | 3445 |

## Physical storage

- How to store the data below:

| | | | | |
|---|---|---|---|---|
| 5 | 0 | 0 | 34 | -1 |
| 0 | 0 | 0 | 13 | 0 |
| -9 | 0 | 0 | 0 | 2 |
| 1 | 0 | 0 | 0 | 0 |
| 0 | -1 | 0 | 0 | 2 |

# Physical storage

- How to store the data below:

| Row | Column | Value |
|---|---|---|
| 1 | 1 | 5 |
| 1 | 4 | 34 |
| 1 | 5 | -1 |
| 2 | 4 | 13 |
| 3 | 1 | -9 |
| 3 | 5 | 2 |
| 4 | 1 | 1 |
| 5 | 2 | -1 |
| 5 | 5 | 2 |

# Physical storage

- Row-based,
- Column-based,
- Key-values stores,
- Multi-dimensional arrays,
- Dense vs. sparse structures,
- Relational OLAP vs. Multidimensional OLAP.

# Physical storage

- The following table can be stored in different ways:

| Year | Products | Sales |
|---|---|---|
| 2010 | Mountain | 5076 |
| 2010 | Road | 4005 |
| 2010 | Touring | 3560 |
| 2011 | Mountain | 6503 |
| 2011 | Road | 4503 |
| 2011 | Touring | 3445 |

## Physical storage

- Row-based storage:

  **001**: 2010, Mountain, 5076, **002**: 2010, Road, 4005, **003**: 2010, Touring, 3560, **004**: 2011, Mountain, 6503, **005**: 2011, Road, 4503 **006**: 2011, Touring, 3445.

## Physical storage

- Row-based storage:

  **001**: 2010, Mountain, 5076, **002**: 2010, Road, 4005, **003**: 2010, Touring, 3560, **004**: 2011, Mountain, 6503, **005**: 2011, Road, 4503 **006**: 2011, Touring, 3445.

- Column-based storage:

  **Y**: 2010, 2010, 2010, 2011, 2011, 2011, **P**: Mountain, Road, Touring, Mountain, Road, Touring, **S**: 5076, 5004, 3560, 6503, 4503, 3445.

  or

  **Y**: 2010: **001**, **002**, **003**, 2011: **004**, **005**, **006**, **P**: Mountain: **001**, **004**, Road: **002**, **005**, Touring: **003**, **006**, **S**: 5076: **001**, 4005, **002**, 3560: **003**, 6503: **004**, 4503: **005**, 3445: **006**

# Physical storage

- Key-value pairs:

  **001,Y**: 2010, **002,Y**: 2010, **003,Y**: 2010, **004,Y**: 2011, **005,Y**: 2011, **006,Y**: 2011, **001,P**: Mountain, **002,P**: Road, **003,P**: Touring, **004,P**: Mountain, **005,P**: Road, **006,P**: Touring, **001,S**: 5076, **002,S**: 4005, **003,S**: 3506, **004,S**: 6503, **005,S**: 4503, **006,S**: 3445

# Physical storage

- Key-value pairs:

  **001,Y**: 2010, **002,Y**: 2010, **003,Y**: 2010, **004,Y**: 2011, **005,Y**: 2011, **006,Y**: 2011, **001,P**: Mountain, **002,P**: Road, **003,P**: Touring, **004,P**: Mountain, **005,P**: Road, **006,P**: Touring, **001,S**: 5076, **002,S**: 4005, **003,S**: 3506, **004,S**: 6503, **005,S**: 4503, **006,S**: 3445

- Multidimensional array:

  **Y**: 2010, 2011, **P**: Mountain, Road, Touring, **S**: 5076, 4005, 3560, 6503, 4503, 3445

# Data access

- Hashing
- Sorting ($\rightarrow$ tree-based indexing).

# Hashing

- Hashing
  - The basic idea behind dictionaries.
  - Extensively used also in many other applications.
  - Some of them will be covered in this lecture.

# Hashing

- Dictionary:
  - Can be implemented as a direct access table, i.e. a large array indexed by a natural key:
    - Keys must be nonnegative integers.
    - The range of keys can be enormous.
  - There are two solutions for these problems: prehash and hash functions.

# Hashing

- Prehash:
  - Maps natural keys to integers
  - Since keys are finite or at least countable, they can be mapped to integers.
  - Implemented in many languages (`hash` or `hashCode` functions).
  - In theory: $x = y \Leftrightarrow \text{hash}(x) = \text{hash}(y)$
  - Keys should not change over time in a given application.

# Hashing

- Prehash for complex objects:

# Hashing

- Prehash for complex objects:
  - In a sense, all data types have values that are composed of bits, and sequences of bits can always be interpreted as integers.

# Hashing

- Prehash for complex objects:
  - In a sense, all data types have values that are composed of bits, and sequences of bits can always be interpreted as integers.
  - String $\Rightarrow$ small integer: convert each character to its ASCII or Unicode equivalent and interpret as an integer; aggregate the integers together.

## Hashing

- Prehash for complex objects:
  - In a sense, all data types have values that are composed of bits, and sequences of bits can always be interpreted as integers.
  - String $\Rightarrow$ small integer: convert each character to its ASCII or Unicode equivalent and interpret as an integer; aggregate the integers together.
  - String $\Rightarrow$ large integer: group the string into a disjoint groups of consecutive integers; concatenate characters in the groups and treat them as single integers; aggregate the integers together.

# Hashing

- Prehash for complex objects:
  - In a sense, all data types have values that are composed of bits, and sequences of bits can always be interpreted as integers.
  - String $\Rightarrow$ small integer: convert each character to its ASCII or Unicode equivalent and interpret as an integer; aggregate the integers together.
  - String $\Rightarrow$ large integer: group the string into a disjoint groups of consecutive integers; concatenate characters in the groups and treat them as single integers; aggregate the integers together.
  - Arrays $\Rightarrow$ map each element of an array to integer; aggregate the integers or groups of them.

# Hashing

- Prehash for complex objects:
  - In a sense, all data types have values that are composed of bits, and sequences of bits can always be interpreted as integers.
  - String $\Rightarrow$ small integer: convert each character to its ASCII or Unicode equivalent and interpret as an integer; aggregate the integers together.
  - String $\Rightarrow$ large integer: group the string into a disjoint groups of consecutive integers; concatenate characters in the groups and treat them as single integers; aggregate the integers together.
  - Arrays $\Rightarrow$ map each element of an array to integer; aggregate the integers or groups of them.
  - Tuples $\Rightarrow$ map each element of a tuple to integer; aggregate the integers or groups of them.

```
S = s                  #Initialize the state.
for k in range(0,m):   #Scan the input data units:
    S = F(S, b[k])     #Combine data unit k into the state.
return S
```

# Hashing

- Hashing:

# Hashing

- Hashing:
  - Reduce universe $\mathcal{U}$ of all integer keys (the result of prehash) down to reasonable size $m$ for table.

# Hashing

- Hashing:
  - Reduce universe $\mathcal{U}$ of all integer keys (the result of prehash) down to reasonable size $m$ for table.
  - Ideally, the number of $n$ elements to be stored equals $m$.

# Hashing

- Hashing:
  - Reduce universe $\mathcal{U}$ of all integer keys (the result of prehash) down to reasonable size $m$ for table.
  - Ideally, the number of $n$ elements to be stored equals $m$.
  - Hash function: $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$

# Hashing

- Hashing:
  - Reduce universe $\mathcal{U}$ of all integer keys (the result of prehash) down to reasonable size $m$ for table.
  - Ideally, the number of $n$ elements to be stored equals $m$.
  - Hash function: $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$
  - Two keys $k_i \neq k_j$ collide if $h(k_i) = h(k_j)$

# Hashing

- Hashing:
  - Reduce universe $\mathcal{U}$ of all integer keys (the result of prehash) down to reasonable size $m$ for table.
  - Ideally, the number of $n$ elements to be stored equals $m$.
  - Hash function: $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$
  - Two keys $k_i \neq k_j$ collide if $h(k_i) = h(k_j)$
  - We want to have good hashing functions (**simple uniform hashing**): each key is equally likely to be hashed to any slot of table independent of where other keys are hashed.

# Hashing

- Hashing:
  - Reduce universe $\mathcal{U}$ of all integer keys (the result of prehash) down to reasonable size $m$ for table.
  - Ideally, the number of $n$ elements to be stored equals $m$.
  - Hash function: $h : \mathcal{U} \rightarrow \{0, 1, \ldots, m-1\}$
  - Two keys $k_i \neq k_j$ collide if $h(k_i) = h(k_j)$
  - We want to have good hashing functions (**simple uniform hashing**): each key is equally likely to be hashed to any slot of table independent of where other keys are hashed.
  - With good hashing functions dictionaries work in $\mathcal{O}(n)$ time.

# Hashing

- Hashing:
  - Reduce universe $\mathcal{U}$ of all integer keys (the result of prehash) down to reasonable size $m$ for table.
  - Ideally, the number of $n$ elements to be stored equals $m$.
  - Hash function: $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$
  - Two keys $k_i \neq k_j$ collide if $h(k_i) = h(k_j)$
  - We want to have good hashing functions (**simple uniform hashing**): each key is equally likely to be hashed to any slot of table independent of where other keys are hashed.
  - With good hashing functions dictionaries work in $\mathcal{O}(n)$ time.
  - Dictionaries to work well need additional elements (e.g., table resizing).

# Hashing

- Hashing:
  - Reduce universe $\mathcal{U}$ of all integer keys (the result of prehash) down to reasonable size $m$ for table.
  - Ideally, the number of $n$ elements to be stored equals $m$.
  - Hash function: $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$
  - Two keys $k_i \neq k_j$ collide if $h(k_i) = h(k_j)$
  - We want to have good hashing functions (**simple uniform hashing**): each key is equally likely to be hashed to any slot of table independent of where other keys are hashed.
  - With good hashing functions dictionaries work in $\mathcal{O}(n)$ time.
  - Dictionaries to work well need additional elements (e.g., table resizing).
  - For the current lecture, we focus on good hashing functions (in many cases we will ignore or allow conflicts).

# Hash functions

- **Division method**:

$$h(k) = k \mod m\,,$$

where $m$ usually is a *prime* number. If it is power of 2 or 10, then the hash are low bits or digits.

# Hash functions

- **Division method**:

$$h(k) = k \mod m\,,$$

where $m$ usually is a *prime* number. If it is power of 2 or 10, then the hash are low bits or digits.

- **Multiplication method**:

$$h(k) = [(a \cdot k) \mod 2^w] \gg (w - r)\,,$$

where $a$ is random, $k$ is $w$ bits, and $m = 2^r$

# Hash functions

- **Division method**:

$$h(k) = k \mod m\,,$$

where $m$ usually is a *prime* number. If it is power of 2 or 10, then the hash are low bits or digits.

- **Multiplication method**:

$$h(k) = [(a \cdot k) \mod 2^w] \gg (w - r)\,,$$

where $a$ is random, $k$ is $w$ bits, and $m = 2^r$

- **Universal hashing**:

$$h(k) = [ax + b \mod p] \mod m\,.$$

where $p > |\mathcal{U}|$ is prime, $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$. This function satisfies

$$P_{h \in H}(h(k_1) = h(k_2)) \leq \frac{1}{m}\,,$$

for each pair of keys $k_1 \neq k_2 \in \{\mathcal{U}\}$.

# Sorting

- For sorted data we can:

## Sorting

- For sorted data we can:
  - Perform binary search:

```
l, r, m = 0, len(t)-1, -1
while l <= r:
  m = l + (r - l)//2
  if t[m] == v:
    break
  elif t[m] < v:
    l = m + 1
  else:
    r = m - 1
return m
```

# Sorting

- For sorted data we can:
  - Perform binary search:

```
l, r, m = 0, len(t)-1, -1
while l <= r:
  m = l + (r - l)//2
  if t[m] == v:
    break
  elif t[m] < v:
    l = m + 1
  else:
    r = m - 1
return m
```

  - Speed-up operations such as group-by or join,

## Sorting

- For sorted data we can:
  - Perform binary search:

```
l, r, m = 0, len(t)-1, -1
while l <= r:
  m = l + (r - l)//2
  if t[m] == v:
    break
  elif t[m] < v:
    l = m + 1
  else:
    r = m - 1
return m
```

  - Speed-up operations such as group-by or join,
  - Build a tree-based index.

## Grouping

- **Group-by** is usually performed in the following way:

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,
    - Sort or re-hash within each bucket to resolve collisions.

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,
    - Sort or re-hash within each bucket to resolve collisions.
- Aggregation functions

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,
    - Sort or re-hash within each bucket to resolve collisions.
- Aggregation functions
  - distributive: `count()`, `sum`, `max`, `min`,

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,
    - Sort or re-hash within each bucket to resolve collisions.
- Aggregation functions
  - distributive: `count()`, `sum`, `max`, `min`,
  - algebraic: `ave()`, `stdev`, `var`,

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,
    - Sort or re-hash within each bucket to resolve collisions.
- Aggregation functions
  - distributive: `count()`, `sum`, `max`, `min`,
  - algebraic: `ave()`, `stdev`, `var`,
  - holistic: `median`, `rank`, `mode`, `distinct count`.

# Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,
    - Sort or re-hash within each bucket to resolve collisions.
- Aggregation functions
  - distributive: `count()`, `sum`, `max`, `min`,
  - algebraic: `ave()`, `stdev`, `var`,
  - holistic: `median`, `rank`, `mode`, `distinct count`.
- Use intermediate results to compute more general group-bys ($\Rightarrow$ Materialization).

# Grouping

- **Example**: Grouping by sorting (Month, City):

| Month | City | Sale |
|---|---|---|
| March | Poznań | 105 |
| March | Warszawa | 135 |
| March | Poznań | 50 |
| May | Warszawa | 100 |
| April | Poznań | 150 |
| April | Kraków | 175 |
| May | Poznań | 70 |
| May | Warszawa | 75 |

# Grouping

- **Example**: Grouping by sorting (Month, City):

| Month | City | Sale |
|---|---|---|
| March | Poznań | 105 |
| March | Warszawa | 135 |
| March | Poznań | 50 |
| May | Warszawa | 100 |
| April | Poznań | 150 |
| April | Kraków | 175 |
| May | Poznań | 70 |
| May | Warszawa | 75 |

$\longrightarrow$

| Month | City | Sale |
|---|---|---|
| March | Poznań | 105 |
| March | Poznań | 50 |
| March | Warszawa | 135 |
| April | Poznań | 150 |
| April | Kraków | 175 |
| May | Poznań | 70 |
| May | Warszawa | 75 |
| May | Warszawa | 100 |

# Grouping

- **Example**: Grouping by sorting (Month, City):

| Month | City | Sale |
|---|---|---|
| March | Poznań | 105 |
| March | Warszawa | 135 |
| March | Poznań | 50 |
| May | Warszawa | 100 |
| April | Poznań | 150 |
| April | Kraków | 175 |
| May | Poznań | 70 |
| May | Warszawa | 75 |

$\longrightarrow$

| Month | City | Sale |
|---|---|---|
| March | Poznań | 105 |
| March | Poznań | 50 |
| March | Warszawa | 135 |
| April | Poznań | 150 |
| April | Kraków | 175 |
| May | Poznań | 70 |
| May | Warszawa | 75 |
| May | Warszawa | 100 |

$\downarrow$

| Month | City | Sale |
|---|---|---|
| March | Poznań | 155 |
| March | Warszawa | 135 |
| April | Poznań | 150 |
| April | Kraków | 175 |
| May | Poznań | 70 |
| May | Warszawa | 175 |

# Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.

# Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.
- An index is a “thin” copy of a relation (not all columns from the relation are included, the index is sorted in a particular way).

# Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.
- An index is a “thin” copy of a relation (not all columns from the relation are included, the index is sorted in a particular way).
- Index-only plans use small indexes in place of large relations.

## Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.
- An index is a "thin" copy of a relation (not all columns from the relation are included, the index is sorted in a particular way).
- Index-only plans use small indexes in place of large relations.
- Query processing on indexes – without accessing base tables.

# Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.
- An index is a “thin” copy of a relation (not all columns from the relation are included, the index is sorted in a particular way).
- Index-only plans use small indexes in place of large relations.
- Query processing on indexes – without accessing base tables.
- Indexes on two and more columns.

# Indexes

- Inverted lists,
- Trees,
- Bitmap index,
- Bit-sliced index,
- Projection index,
- Join index.

# Inverted list

- **Inverted list** stores a mapping from content (e.g., words) to its locations in a database (e.g., in documents):

| |
|---|
| document 1 $\longrightarrow$ word 1, word 5, word 4, word 175, word 7 |
| document 2 $\longrightarrow$ word 54, word 1, word 4, word 6, word 71 |
| document 3 $\longrightarrow$ word 5, word 175, word 11 |
| ... |

- **Inverted list** stores a mapping from content (e.g., words) to its locations in a database (e.g., in documents):

$$
\begin{array}{c}
\text{word } 1 \longrightarrow \text{document } 1\text{, document } 2 \\
\ldots \\
\text{word } 4 \longrightarrow \text{document } 1\text{, document } 2 \\
\text{word } 5 \longrightarrow \text{document } 1\text{, document } 3 \\
\text{word } 6 \longrightarrow \text{document } 2\text{, } \ldots \\
\ldots
\end{array}
$$

# Bitmap index

- Bitmap indexes use bit arrays (commonly called "bitmaps") to encode values on a given attribute and answer queries by performing bitwise logical operations on these bitmaps.

# Bitmap index

- Bitmap indexes use bit arrays (commonly called "bitmaps") to encode values on a given attribute and answer queries by performing bitwise logical operations on these bitmaps.

| Customer | City | Car |
|---|---|---|
| C1 | Detroit | Ford |
| C2 | Chicago | Honda |
| C3 | Detroit | Honda |
| C4 | Poznań | Ford |
| C5 | Paris | BMW |
| C6 | Paris | Nissan |

# Bitmap index

- Bitmap indexes use bit arrays (commonly called "bitmaps") to encode values on a given attribute and answer queries by performing bitwise logical operations on these bitmaps.

| Customer | City | Car |
|---|---|---|
| C1 | Detroit | Ford |
| C2 | Chicago | Honda |
| C3 | Detroit | Honda |
| C4 | Poznań | Ford |
| C5 | Paris | BMW |
| C6 | Paris | Nissan |

↓

| Customer | Chicago | Detroit | Paris | Poznań |
|---|---|---|---|---|
| C1 | 0 | 1 | 0 | 0 |
| C2 | 1 | 0 | 0 | 0 |
| C3 | 0 | 1 | 0 | 0 |
| C4 | 0 | 0 | 0 | 1 |
| C5 | 0 | 0 | 1 | 0 |
| C6 | 0 | 0 | 1 | 0 |

→

| Bitmap | Array of bytes |
|---|---|
| Chicago | 010000 (00) |
| Detroit | 101000 (00) |
| Paris | 010011 (00) |
| Poznań | 000100 (00) |

# Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),

## Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,

# Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,
- Usually bitmap indexes are compressed,

## Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,
- Usually bitmap indexes are compressed,
- Works poorly for high cardinality domains since the number of bitmaps increases,

# Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,
- Usually bitmap indexes are compressed,
- Works poorly for high cardinality domains since the number of bitmaps increases,
- Difficult to maintain – need reorganization when relation sizes change (new bitmaps)

## Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,
- Usually bitmap indexes are compressed,
- Works poorly for high cardinality domains since the number of bitmaps increases,
- Difficult to maintain – need reorganization when relation sizes change (new bitmaps)
- Can be used with other index structures (e.g., tree-based indexes).

# Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical** (**integer**) **attributes**:

# Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical** (**integer**) **attributes**:
  - Efficient aggregation,

# Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical** (**integer**) **attributes**:
  - Efficient aggregation,
  - Efficient range filtering.

# Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical** (**integer**) **attributes**:
  - Efficient aggregation,
  - Efficient range filtering.
- **Definition**:

# Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical (integer) attributes**:
  - Efficient aggregation,
  - Efficient range filtering.
- **Definition**:
  - Assume, that values of attribute $a$ are integer numbers coded by $n+1$ bits. In this case, attribute $a$ can be stored as binary attributes $a_0, a_1, \ldots, a_n$, such that

    $$a = \sum_{i=0}^{n} 2^i a_i = a_0 + 2a_1 + 2^2 a_2 \cdots + 2^n a_n.$$

    Each binary attribute $a_i$ can be stored as bitmap index. Set of bitmap indexes of $a_i$, $i = 0, \ldots, n$, is the **bit-sliced index**.

# Bit-sliced index

- **Example**:

| Amount | Bitmap |
|---|---|
| 5 | 01**0**1 |
| 13 | 11**0**1 |
| 2 | 00**1**0 |
| 6 | 01**1**0 |
| 7 | 01**1**1 |

**Bit-sliced index**:

- B4: 01000
- B3: 11011
- B2: **00111**
- B1: 11001

# Bit-sliced index

- **Example**:
  - Computing the sum:

| Amount |
|---|
| 5 |
| 13 |
| 2 |
| 6 |
| 7 |
| Sum: 33 |

| **Bit-sliced index**: | **Counting ones**: |
|---|---|
| B4: 01000 | 1 |
| B3: 11011 | 4 |
| B2: 00111 | 3 |
| B1: 11001 | 3 |

Final results: $1 \cdot 2^3 + 4 \cdot 2^2 + 3 \cdot 2^1 + 3 \cdot 2^0 = 8 + 16 + 6 + 3 = 33$

- **Example**:
  - Computing the sum:

| Amount |
|---|
| 5 |
| 13 |
| 2 |
| 6 |
| 7 |
| Sum: 33 |

**Bit-sliced index**:
B4: 01000
B3: 11011
B2: 00111
B1: 11001

**Counting ones**:
1
4
3
3

Final results: $1 \cdot 2^3 + 4 \cdot 2^2 + 3 \cdot 2^1 + 3 \cdot 2^0 = 8 + 16 + 6 + 3 = 33$

**Problem**: How to efficiently count the number of ones in a bitmap?

# Fast bitmap count

- Count the number of 1's in a bitmap:

## Fast bitmap count

- Count the number of 1's in a bitmap:
  - Treat the bitmap as a byte array.
  - Pre-compute lookup table with number of 1's in each byte.
  - Cycle through bitmap one byte at a time, accumulating count using. lookup table

## Fast bitmap count

- Count the number of 1's in a bitmap:
  - Treat the bitmap as a byte array.
  - Pre-compute lookup table with number of 1's in each byte.
  - Cycle through bitmap one byte at a time, accumulating count using. lookup table
- **Pseudocode**:

```
numSetBits[0] = 0;
numSetBits[1] = 1;
numSetBits[2] = 1;
numSetBits[3] = 2;
...
numSetBits[255] = 8;
count = 0;
for (int i = 0; i < n/8; i++)
     count += numSetBits[bitmap[i]];
```

# Fast bitmap count

- Count the number of 1's in a bitmap:
  - Treat the bitmap as a byte array.
  - Pre-compute lookup table with number of 1's in each byte.
  - Cycle through bitmap one byte at a time, accumulating count using. lookup table
- **Pseudocode**:

```
numSetBits[0] = 0;
numSetBits[1] = 1;
numSetBits[2] = 1;
numSetBits[3] = 2;
...
numSetBits[255] = 8;
count = 0;
for (int i = 0; i < n/8; i++)
     count += numSetBits[bitmap[i]];
```

- Treating bitmap as short int array $\rightarrow$ even faster
  - Lookup table has 65536 entries instead of 256.
  - Bitmap of $n$ bits $\rightarrow$ only add $n/16$ numbers.

# Fast bitmap count

- Count the number of 1's in a bitmap
  - Use smartly properties of binary coding.
  - Making count to be linear with the number of ones.

# Fast bitmap count

- Count the number of 1's in a bitmap
  - Use smartly properties of binary coding.
  - Making count to be linear with the number of ones.
- **Pseudocode**

```
word = bitmap[i];
count = 0;
while (word != 0)
     word &= (word - 1);
     count++;
```

## Storing and accessing multidimensional cubes

- Dense and sparse dimensions
- Organize a multi-dimensional cube by properly setting dimension types.

## Storing and accessing multidimensional cubes

- Dense and sparse dimensions
- Organize a multi-dimensional cube by properly setting dimension types.
- **Example**: Assume 3 dimensions, like Product, Localization, Date and several measures like Revenue, Expenses, Netto, etc.
  - Date and measures are rather dense,
  - Product and Localization are rather sparse.
  - Two extreme data cube organizations are possible.

## Storing and accessing multidimensional cubes

- **Example**: Assume 3 dimensions, like Product, Localization, Date and several measures like Revenue, Expenses, Netto, etc.
  - Two extreme data cube organizations are possible.

| | | JAN | | | FEB | | | MAR | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | East | West | South | East | West | South | East | West | South |
| | Prod. A | | XXX | XXX | | XXX | XXX | | XXX | XXX |
| Rev. | Prod. B | XXX | XXX | | XXX | XXX | | XXX | XXX | |
| | Prod. C | XXX | XXX | | XXX | XXX | | XXX | XXX | |
| | Prod. A | | XXX | XXX | | XXX | XXX | | XXX | XXX |
| Exp. | Prod. B | XXX | XXX | | XXX | XXX | | XXX | XXX | |
| | Prod. C | XXX | XXX | | XXX | XXX | | XXX | XXX | |
| | Prod. A | | XXX | XXX | | XXX | XXX | | XXX | XXX |
| Net. | Prod. B | XXX | XXX | | XXX | XXX | | XXX | XXX | |
| | Prod. C | XXX | XXX | | XXX | XXX | | XXX | XXX | |

## Storing and accessing multidimensional cubes

- **Example**: Assume 3 dimensions, like Product, Localization, Date and several measures like Revenue, Expenses, Netto, etc.
  - Two extreme data cube organizations are possible.

| | | East | | | West | | | South | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | JAN | FEB | MAR | JAN | FEB | MAR | JAN | FEB | MAR |
| | Rev. | | | | XXX | XXX | XXX | XXX | XXX | XXX |
| Prod. A | Exp. | | | | XXX | XXX | XXX | XXX | XXX | XXX |
| | Net. | | | | XXX | XXX | XXX | XXX | XXX | XXX |
| | Rev. | XXX | XXX | XXX | XXX | XXX | XXX | | | |
| Prod. B. | Exp. | XXX | XXX | XXX | XXX | XXX | XXX | | | |
| | Net. | XXX | XXX | XXX | XXX | XXX | XXX | | | |
| | Rev. | XXX | XXX | XXX | XXX | XXX | XXX | | | |
| Prod. C. | Exp. | XXX | XXX | XXX | XXX | XXX | XXX | | | |
| | Net. | XXX | XXX | XXX | XXX | XXX | XXX | | | |

# Storing and accessing multidimensional cubes

- **Example**: Assume 3 dimensions, like Product, Localization, Date and several measures like Revenue, Expenses, Netto, etc.
  - Two extreme data cube organizations are possible.
    - The first organization is inefficient.
    - The second organization allows to efficiently store the cube using $3 \times 3$ data chunks — some of the chunks are empty.

## Storing and accessing multidimensional cubes

- Construct an index on sparse dimensions.

# Storing and accessing multidimensional cubes

- Construct an index on sparse dimensions.
- Each leaf points to a multidimensional array that stores dense dimensions.

# Storing and accessing multidimensional cubes

- Construct an index on sparse dimensions.
- Each leaf points to a multidimensional array that stores dense dimensions.
- The multidimensional arrays can be still compressed: bitmap compression, run-length encoding, etc.

- **Example**:
  - A sparse array:

| | Product | Mountain | Road | Touring |
|---|---|---|---|---|
| Day | 1/1/2010 | | | 3 |
| | 2/1/2011 | | 2 | |
| | 3/1/2011 | | | 5 |

can be stored as a sequence of non-missing values

$$3, 2, 5$$

## Compression

- **Example**:
  - A sparse array:

| | Product | Mountain | Road | Touring |
|---|---|---|---|---|
| Day | 1/1/2010 | | | 3 |
| | 2/1/2011 | | 2 | |
| | 3/1/2011 | | | 5 |

    can be stored as a sequence of non-missing values

$$3, 2, 5,$$

    but we need add additional information about positions of these values:

# Compression

- **Example**:
  - A sparse array:

| | Product | Mountain | Road | Touring |
|---|---|---|---|---|
| Day | 1/1/2010 | | | 3 |
| | 2/1/2011 | | 2 | |
| | 3/1/2011 | | | 5 |

can be stored as a sequence of non-missing values

$$3, 2, 5,$$

but we need add additional information about positions of these values:

- Indexes: 3,5,9
- Gaps: 2,1,3
- Bitmaps: 001010001
- Run-length codes: Null, Null, 3, Null, 2, Null$\times$3, 5
- Indexes and gaps can be further coded by prefix codes.

# Outline


1 Physical storage and data access

2 Materialization

3 Summary

# Materialization

- Relational and multidimensional model with summarizations:

| Year | Products | Sales |
|---|---|---|
| 2010 | Mountain | 5076 |
| 2010 | Road | 4005 |
| 2010 | Touring | 3560 |
| 2011 | Mountain | 6503 |
| 2011 | Road | 4503 |
| 2011 | Touring | 3445 |
| 2010 | * | 12461 |
| 2011 | * | 14451 |
| * | Mountain | 11579 |
| * | Road | 6503 |
| * | Touring | 7005 |
| * | * | 27092 |

| | Product | Mountain | Road | Touring | All |
|---|---|---|---|---|---|
| Year | 2010 | 5076 | 4005 | 3560 | 12641 |
| | 2011 | 6503 | 4503 | 3445 | 14451 |
| | All | 11579 | 8508 | 7005 | 27092 |

# Materialization

- Trade-off between query performance and load performance
- To improve performance of query processing:
  - Precompute as much as possible
  - Build additional data structures like indexes
- The costs of the above are:
  - Disk space,
  - Load time,
  - Processing time of building and updating of data structures

## Materialization

- Typical techniques:

# Materialization

- Typical techniques:
  - Materialized views or indexed views.

# Materialization

- Typical techniques:
  - Materialized views or indexed views.
  - Subcubes or aggregations.

# Materialization

- Typical techniques:
  - Materialized views or indexed views.
  - Subcubes or aggregations.
- Aggregates should be computed from previously computed aggregates, rather than from the base fact table.

# Materialization

- Typical techniques:
  - Materialized views or indexed views.
  - Subcubes or aggregations.
- Aggregates should be computed from previously computed aggregates, rather than from the base fact table.
- The problem appears with maintenance of the materialized views: recomputation and incremental updating.

# View vs. materialized views

- **View** is a derived relation defined in terms of base (stored) relations.
- **Materialized view** (or indexed view) is a view stored in a database that is updated from the original base tables from time to time.

# Query re-write

- **Query rewrite**: transforms a given query expressed in terms of base tables or views into a statement accessing one or more materialized views (e.g., aggregates) that are defined on the detail tables.
- The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the query.

# Query re-write

- **Example**: Materialized views in SQL
  - Materialized view $V$:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product id = p.id AND p.year_of_release > 2010
GROUP BY p.name, p.year_of_release;
```

  - Materialized view $V$ consists of:
    - Join of the fact table with dimension table,
    - Group by dimension attributes,
    - Aggregation of measures included in fact table.

# Query re-write

- **Example**: Materialized views in SQL

# Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product id = p.id AND p.year_of_release > 2011
GROUP BY p.name, p.year of release;
```

# Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product id = p.id AND p.year_of_release > 2011
GROUP BY p.name, p.year of release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year of release > 2011;
```

# Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product id = p.id AND p.year_of_release > 2011
GROUP BY p.name, p.year of release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year of release > 2011;
```

  - The query re-write is possible since the exact match holds:

# Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

    ```
    SELECT p.name, p.year_of_release, sum(s.price) as price
    FROM Sales s, Product p
    WHERE s.product id = p.id AND p.year_of_release > 2011
    GROUP BY p.name, p.year of release;
    ```
  - Query rewrite

    ```
    SELECT p.name, p.year_of_release, price
    FROM V
    WHERE year of release > 2011;
    ```
  - The query re-write is possible since the exact match holds:
    - all the projected columns are also in $V$,

# Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product id = p.id AND p.year_of_release > 2011
GROUP BY p.name, p.year of release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year of release > 2011;
```

  - The query re-write is possible since the exact match holds:
    - all the projected columns are also in $V$,
    - the same aggregate functions are used on all measures,

# Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product id = p.id AND p.year_of_release > 2011
GROUP BY p.name, p.year of release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year of release > 2011;
```

  - The query re-write is possible since the exact match holds:
    - all the projected columns are also in $V$,
    - the same aggregate functions are used on all measures,
    - all selection conditions in the query imply the selection conditions in $V$,

# Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product id = p.id AND p.year_of_release > 2011
GROUP BY p.name, p.year of release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year of release > 2011;
```

  - The query re-write is possible since the exact match holds:
    - all the projected columns are also in $V$,
    - the same aggregate functions are used on all measures,
    - all selection conditions in the query imply the selection conditions in $V$,
    - the attributes present in selection conditions that are strictly stronger than selection conditions defined in $V$, are also present in $V$.

# Exercise

- There exists a materialized view denoted by $V$:

```
SELECT name, model, year,
sum(price) as price, count(*) as card
FROM Sales NATURAL JOIN Cars
GROUP BY name, model, year;
```

  How does the query re-write work for the query below?

```
SELECT name, model, avg(price)
FROM Sales NATURAL JOIN Cars
WHERE year > 2010 GROUP BY name, model;
```

# Maintenance of materialized views

- Let $V$ be the materialized view defined by a query $Q$ over a set $R$ of relations

$$V = Q(R)\,.$$

## Maintenance of materialized views

- Let $V$ be the materialized view defined by a query $Q$ over a set $R$ of relations
$$V = Q(R)\,.$$
- When the relations in $R$ are updated, then $V$ becomes inconsistent.

## Maintenance of materialized views

- Let $V$ be the materialized view defined by a query $Q$ over a set $R$ of relations
$$V = Q(R)\,.$$
- When the relations in $R$ are updated, then $V$ becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between $R$ and $V$.

# Maintenance of materialized views

- Let $V$ be the materialized view defined by a query $Q$ over a set $R$ of relations
$$V = Q(R)\,.$$
- When the relations in $R$ are updated, then $V$ becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between $R$ and $V$.
- Different aspects:

## Maintenance of materialized views

- Let $V$ be the materialized view defined by a query $Q$ over a set $R$ of relations
$$V = Q(R)\,.$$
- When the relations in $R$ are updated, then $V$ becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between $R$ and $V$.
- Different aspects:
  - Immediate and delayed refresh.

## Maintenance of materialized views

- Let $V$ be the materialized view defined by a query $Q$ over a set $R$ of relations
$$V = Q(R)\,.$$
- When the relations in $R$ are updated, then $V$ becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between $R$ and $V$.
- Different aspects:
    - Immediate and delayed refresh.
    - Full refresh and view maintenance.

## Maintenance of materialized views

- Let $V$ be the materialized view defined by a query $Q$ over a set $R$ of relations
$$V = Q(R)\,.$$
- When the relations in $R$ are updated, then $V$ becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between $R$ and $V$.
- Different aspects:
  - Immediate and delayed refresh.
  - Full refresh and view maintenance.
  - Maintainable and partially maintainable views.

## Maintenance of materialized views

- Let $V$ be the materialized view defined by a query $Q$ over a set $R$ of relations
$$V = Q(R)\,.$$
- When the relations in $R$ are updated, then $V$ becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between $R$ and $V$.
- Different aspects:
  - Immediate and delayed refresh.
  - Full refresh and view maintenance.
  - Maintainable and partially maintainable views.
- **Example**: How to maintain the materialized view defined below?

```
V = SELECT min(A.a) FROM A
```

# Outline

1 Physical storage and data access

2 Materialization

3 Summary

# Summary

- Physical storage and data access,
- Materialization, denormalization and summarization.