Data Warehousing
Lecture 8

Toon Calders
toon.calders@ulb.ac.be
Summary

• How is the data stored?
  – Relational database (ROLAP)
  – Specialized structures (MOLAP)

• How can we speed up computation?
  – Materialized views
  – Indexing structures
  – Partitioning
How Does it Fit In?

• We know what part of the full cube we want to materialize, and how to store it. We made the problem smaller but did not solve it.

  – Before partial materialization:
    Answer (supplier) from \((\text{part, supplier, customer})\)

  – After partial materialization:
    Answer (supplier) from \((\text{supplier, customer})\)
How Does it Fit In?

• Not all queries are of the type

```sql
SELECT D1, ..., Dk, sum(M)
FROM R
GROUP BY D1, ..., Dk
```
How Does it Fit In?

• Example of another type of query:

```sql
SELECT supplier, year, min(price)
FROM "cube"
WHERE
    product = "toilet paper"
    and (year = 2009 or year = 2010)
GROUP BY supplier, year
```
Indexing Principle

No index
Indexing Principle

No index
Indexing Principle

• Database Equivalent

No index $\rightarrow$ Expensive

Full table scan

Index $\rightarrow$ Inexpensive

index lookup

+ Retrieve data page
Why not Just Use B-Trees?

- The RDBMS work horse!

- Make index on (Country, Category, Brand, Chain)
Why not Just Use B-Trees?

B(+)-Trees no longer suffice

- **Fixed order** between attributes

  Index(A,B,C) on R supports:
  - Selections on A, on AB, on ABC

  Does not support:
  - Selections on B, BC, C

  We need **exponentially many** B-trees to support all possible selections

- **Attributes are spread over different tables**
Summary

• How can we speed up computation?
  – Materialized Views
  – Indexing structures
    • Bitmap index
    • Join index
    • Bitmap-join index
  – Partitioning
Bitmap Index: Definition

• Indexing structure for
  – One attribute $R.A$ for one relation $R$
  – Optimizing queries making Boolean combinations of selections on indexed attributes

• For every value $v$ in the active domain of $A$ store a bitmap
  – Length of the bitmap = size of the relation $R$
  – Bitmap has value 1 on position $k$ if the $k$th tuple of $R$ has value $v$ in attribute $A$
# Bitmap Index: Example

<table>
<thead>
<tr>
<th>Product</th>
<th>Country</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>TV</td>
<td>Ireland</td>
<td>20</td>
</tr>
<tr>
<td>TV</td>
<td>France</td>
<td>126</td>
</tr>
<tr>
<td>HiFi</td>
<td>Germany</td>
<td>56</td>
</tr>
<tr>
<td>PC</td>
<td>Ireland</td>
<td>23</td>
</tr>
<tr>
<td>TV</td>
<td>France</td>
<td>138</td>
</tr>
<tr>
<td>PC</td>
<td>Germany</td>
<td>48</td>
</tr>
</tbody>
</table>

- **Index for Country:**

<table>
<thead>
<tr>
<th>Country</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ireland</td>
<td>100100</td>
</tr>
<tr>
<td>France</td>
<td>010010</td>
</tr>
<tr>
<td>Germany</td>
<td>001001</td>
</tr>
</tbody>
</table>
## Bitmap Index: Example

<table>
<thead>
<tr>
<th>Product</th>
<th>Country</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>TV</td>
<td>Ireland</td>
<td>20</td>
</tr>
<tr>
<td>TV</td>
<td>France</td>
<td>126</td>
</tr>
<tr>
<td>HiFi</td>
<td>Germany</td>
<td>56</td>
</tr>
<tr>
<td>PC</td>
<td>Ireland</td>
<td>23</td>
</tr>
<tr>
<td>TV</td>
<td>France</td>
<td>138</td>
</tr>
<tr>
<td>PC</td>
<td>Germany</td>
<td>48</td>
</tr>
</tbody>
</table>

### Index for Country:
- **Ireland**: 100100
- **France**: 010010
- **Germany**: 001001

### Index for Product:
- **TV**: 110010
- **HiFi**: 001000
- **PC**: 000101
SELECT sum(Sales) 
FROM PCS 
WHERE (Country = Ireland or Country = France) 
and not(Product = TV) 

Access only tuples corresponding to a 1 in the bitmap: 
( 100100 | 010010 ) & ! 110010 = 000100
Bitmap Index: Space

• Size of bitmaps can be reduced
  E.g., run-length encoding (RLE):
  1111000111100000001111000 is encoded as
  4x1;3x0;4x1;7x0;4x1;3x0
  – Reduces storage requirements significantly
  – Logical operations can work directly on RLE

• BTree maps values to the (variable length) bitmaps
Working Directly on RLE

- **Brussels:** 000000011000010000000
  7x0, 2x1, 4x0, 1x1, 6x0

- **TV:** 00010001110010000000
  3x0, 1x1, 3x0, 3x1, 2x0, 1x1, 7x0

- **Orange:** 00000111110101110000
  5x0, 5x1, 1x0, 1x1, 1x0, 3x1, 4x0

- !(orange) & (Brussels | TV):
  - !orange: 5x1, 5x0, 1x1, 1x0, 1x1, 3x0, 4x1
  - Brussels | TV: 3x0, 1x1, 3x0, 3x1, 2x0, 2x1, 6x0
  - !o & B|T: 3x0, 1x1, 8x0, 1x1, 7x0
Bitmap Index

B-tree

internal node

Belgium
- 4x0
- 2x1
- 3x0

Greece
- 2x1
- 3x0
- 1x1
- 2x0

Romania
- 2x0
- 2x1
- 4x0

Spain
- 6x0
- 2x1

Leafs contain the bitmaps
Variants of Bitmap Index

• Numerical attributes
  – Discretize into intervals
    E.g., temperature into ]-10,-5], ]-5,0], ]0,5], ]5,10], ...
  – Make one bitmap for each bin
  – Combine bitmaps at query time
    \[
    \text{temperature} > 5 \implies \text{b(]0,5])} | \text{b(]5,10])} | \ldots
    \]

• Other option: make overlapping bitmaps
  – Bitmap for all tuples with t>-10, t>-5, t>0, t>5, ...
    \[
    \text{temperature between 0 and 5: } \text{b(t>0)} \& \neg \text{b(t>5)}
    \]
Variants of Bitmap Index

• Bit-sliced index for nominal attribute with many values:
  – Encode values as integers
  – Make bitmap for bit 0, for bit 1, for bit 2, ...

<table>
<thead>
<tr>
<th>Customer</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Antwerp</td>
</tr>
<tr>
<td>C2</td>
<td>Brussels</td>
</tr>
<tr>
<td>C3</td>
<td>Eindhoven</td>
</tr>
<tr>
<td>C4</td>
<td>Eindhoven</td>
</tr>
</tbody>
</table>

Variants of Bitmap Index

- Bit-sliced index for nominal attribute with many values:
  - Encode values as integers
  - Make bitmap for bit 0, for bit 1, for bit 2, ...

<table>
<thead>
<tr>
<th>Customer</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0000</td>
</tr>
<tr>
<td>C2</td>
<td>0001</td>
</tr>
<tr>
<td>C3</td>
<td>1011</td>
</tr>
<tr>
<td>C4</td>
<td>1011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 3</th>
<th>Bit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Variants of Bitmap Index

• Bit-sliced index for nominal attribute with many values:
  – Encode values as integers
  – Make bitmap for bit 0, for bit 1, for bit 2, ...

<table>
<thead>
<tr>
<th>Customer</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0000</td>
</tr>
<tr>
<td>C2</td>
<td>0001</td>
</tr>
<tr>
<td>C3</td>
<td>1011</td>
</tr>
<tr>
<td>C4</td>
<td>1011</td>
</tr>
</tbody>
</table>

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

Antwerp or Brussels  →  ( !b1 & !b2 &!b3 & !b4 ) | ( !b1 & !b2 &!b3 & b4 )  
                       →  ( !b1 & !b2 &!b3)
Bitmap Index: Pros and Cons

• Bitmap Indices are hard to maintain
  – Due to compression
  – Usually rebuilt after bulk update

• Particularly useful when there are multiple bitmap indices

• For low cardinality attributes
  – Bitmaps lose their edge when cardinality is high
Index Combination: Bitmap Filtering

• Not all systems support bitmap indices (e.g., SQL Server)
  – Yet, has a different way to combine indices
    → bitmap filtering

Index on A → list of rowIDs
Index on B → list of rowIDs

-----------
Intersect lists
Index Combination: Bitmap Filtering

• Not all systems support bitmap indices (e.g., SQL Server)
  – Yet, has a different way to combine indices → bitmap filtering

Index on A → list of rowIDs → bitmap
Index on B → list of rowIDs → bitmap

-------------
bitmap arithmetics
(Vertical) Projection Index

- Almost the same
  - Instead of storing bitmaps: project on columns
  - Keep all projections in the same order

<table>
<thead>
<tr>
<th>Customer</th>
<th>Age</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>30</td>
<td>Antwerp</td>
</tr>
<tr>
<td>C2</td>
<td>34</td>
<td>Brussels</td>
</tr>
<tr>
<td>C3</td>
<td>39</td>
<td>Eindhoven</td>
</tr>
<tr>
<td>C4</td>
<td>65</td>
<td>Eindhoven</td>
</tr>
</tbody>
</table>

Added to the database
(Vertical) Projection Index

• Advantage:
  – better data locality if only few attributes are needed
  – No separate, hard to maintain index needed

• Column-store databases have proven to be highly performant for managing large datasets
Remark: Bitmap and Projection Index

• Mapping bitmaps to tuples not always straightforward
  – For instance, where to locate 5th tuple?
• If tuples are stored consecutively, and have equal length: offset+(nr-1)*length
• Otherwise: next to bitmaps array with physical addresses
  – entry i in array holds physical location of ith tuple
  – Only one such array per table needed
Summary

• How can we speed up computation?
  – Materialized Views
  – Indexing structures
    • Bitmap index / Projection index
    • Join index
    • Bitmap-join index
    • Indexing fact and dimension tables
  – Partitioning
Join Index

• Traditional indexes:
  – value of R.A $\rightarrow$ rows/RIDs in R

• Join indices:
  – Value of R.A $\rightarrow$ RIDs in S

• Data warehouse:
  – Attribute in dimension table $\rightarrow$ rows in fact table
  – Join indexes can span multiple dimensions
Join Index: Example

Sales

<table>
<thead>
<tr>
<th>Date</th>
<th>pID</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Jack</td>
</tr>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Pete</td>
</tr>
<tr>
<td>13/5/12</td>
<td>3</td>
<td>John</td>
</tr>
<tr>
<td>14/5/12</td>
<td>2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Products

<table>
<thead>
<tr>
<th>pID</th>
<th>pName</th>
<th>Category</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jacket</td>
<td>Non-food</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Bread</td>
<td>Food</td>
<td>2.1</td>
</tr>
<tr>
<td>3</td>
<td>Beer</td>
<td>Food</td>
<td>1.5</td>
</tr>
<tr>
<td>4</td>
<td>Paper</td>
<td>Non-food</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Date | pID | Client | pName  | Category    | Price |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Jack</td>
<td>Jacket</td>
<td>Non-food</td>
<td>10</td>
</tr>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Pete</td>
<td>Jacket</td>
<td>Non-food</td>
<td>10</td>
</tr>
<tr>
<td>13/5/12</td>
<td>3</td>
<td>John</td>
<td>Beer</td>
<td>Food</td>
<td>1.5</td>
</tr>
<tr>
<td>14/5/12</td>
<td>2</td>
<td>Mary</td>
<td>Bread</td>
<td>Food</td>
<td>2.1</td>
</tr>
</tbody>
</table>
Join Index: Example

**Sales**

<table>
<thead>
<tr>
<th>Date</th>
<th>pID</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Jack</td>
</tr>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Pete</td>
</tr>
<tr>
<td>13/5/12</td>
<td>3</td>
<td>John</td>
</tr>
<tr>
<td>14/5/12</td>
<td>2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

**Products**

<table>
<thead>
<tr>
<th>pID</th>
<th>pName</th>
<th>Category</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jacket</td>
<td>Non-food</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Bread</td>
<td>Food</td>
<td>2.1</td>
</tr>
<tr>
<td>3</td>
<td>Beer</td>
<td>Food</td>
<td>1.5</td>
</tr>
<tr>
<td>4</td>
<td>Paper</td>
<td>Non-food</td>
<td>1.2</td>
</tr>
</tbody>
</table>

**SP_category_jidx**

<table>
<thead>
<tr>
<th>Category</th>
<th>RID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-food</td>
<td>r1,r2</td>
</tr>
<tr>
<td>Food</td>
<td>r3,r4</td>
</tr>
</tbody>
</table>
Join Index: Data Warehouse

- Join index can **index tuples in the fact table based on an attribute in a dimension**

  - E.g., Index tuples in the fact table for the attribute *Country of Customer*

```
<table>
<thead>
<tr>
<th>Customer</th>
<th>FactTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>CID, CName, CAddress, City, Country</td>
<td>CID, SID, PID, DID, Quantity, TotalPrice</td>
</tr>
</tbody>
</table>
```

- E.g., Index tuples in the fact table for the attribute *Country of Customer*
Join Index: Variant

- If index attribute is primary key of one table: directly store RIDs into the table itself
  - Avoids lookup in index when conditions on product.

<table>
<thead>
<tr>
<th>pID</th>
<th>pName</th>
<th>Category</th>
<th>Price</th>
<th>RIDSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jacket</td>
<td>Non-food</td>
<td>10</td>
<td>r1, r2</td>
</tr>
<tr>
<td>2</td>
<td>Bread</td>
<td>Food</td>
<td>2.1</td>
<td>r4</td>
</tr>
<tr>
<td>3</td>
<td>Beer</td>
<td>Food</td>
<td>1.5</td>
<td>r3</td>
</tr>
<tr>
<td>4</td>
<td>Paper</td>
<td>Non-food</td>
<td>1.2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>pID</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Jack</td>
</tr>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Pete</td>
</tr>
<tr>
<td>13/5/12</td>
<td>3</td>
<td>John</td>
</tr>
<tr>
<td>14/5/12</td>
<td>2</td>
<td>Mary</td>
</tr>
</tbody>
</table>
Bitmap-join index

- Logical combination of bitmap index and join index

<table>
<thead>
<tr>
<th>Date</th>
<th>pID</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Jack</td>
</tr>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Pete</td>
</tr>
<tr>
<td>13/5/12</td>
<td>3</td>
<td>John</td>
</tr>
<tr>
<td>14/5/12</td>
<td>2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-food</td>
<td>1100</td>
</tr>
<tr>
<td>Food</td>
<td>0011</td>
</tr>
</tbody>
</table>

Sales

SP_category_bjidx

SP_category_jidx

<table>
<thead>
<tr>
<th>Category</th>
<th>RID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-food</td>
<td>r1,r2</td>
</tr>
<tr>
<td>Food</td>
<td>r3,r4</td>
</tr>
</tbody>
</table>
**Bitmap-Join Index: Example**

<table>
<thead>
<tr>
<th>Date</th>
<th>pID</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Jack</td>
</tr>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Pete</td>
</tr>
<tr>
<td>13/5/12</td>
<td>3</td>
<td>John</td>
</tr>
<tr>
<td>14/5/12</td>
<td>2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

**SP_category_bjidx**

<table>
<thead>
<tr>
<th>Category</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-food</td>
<td>1100</td>
</tr>
<tr>
<td>Food</td>
<td>0011</td>
</tr>
</tbody>
</table>

**SC_city_bjidx**

<table>
<thead>
<tr>
<th>City</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brussels</td>
<td>1001</td>
</tr>
<tr>
<td>Eindhoven</td>
<td>0110</td>
</tr>
</tbody>
</table>

```sql
SELECT date
FROM Sales S JOIN Product P JOIN Customer C
WHERE P.Category = "Food" and C.City = "Brussels";
```
**Bitmap-Join Index: Example**

<table>
<thead>
<tr>
<th>Date</th>
<th>pID</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Jack</td>
</tr>
<tr>
<td>10/5/12</td>
<td>1</td>
<td>Pete</td>
</tr>
<tr>
<td>13/5/12</td>
<td>3</td>
<td>John</td>
</tr>
<tr>
<td><strong>14/5/12</strong></td>
<td><strong>2</strong></td>
<td>Mary</td>
</tr>
</tbody>
</table>

**SELECT** date
**FROM** Sales S **JOIN** Product P **JOIN** Customer C
**WHERE**
P.Category = "Food" and C.City = "Brussels";

**SP_category_bjidx**

<table>
<thead>
<tr>
<th>Category</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-food</td>
<td>1100</td>
</tr>
<tr>
<td><strong>Food</strong></td>
<td><strong>0011</strong></td>
</tr>
</tbody>
</table>

**SC_city_bjidx**

<table>
<thead>
<tr>
<th>City</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Brussels</strong></td>
<td><strong>1001</strong></td>
</tr>
<tr>
<td>Eindhoven</td>
<td>0110</td>
</tr>
</tbody>
</table>

0011 & 1001 \(\rightarrow\) 0001
Indices in Practice

• Several commercial products implement bitmap, join & bitmap-join index
  – E.g., Oracle (EE) offers a compressed bitmap index
    
    \[
    \text{CREATE BITMAP INDEX cust_sales_bji}
    \text{ON sales(customers.city)}
    \text{FROM sales, customers}
    \text{WHERE sales.client = customers.name;}
    \]

• Some products build bitmaps \textit{on the fly}
  – E.g., SQLServer 2008 \(\rightarrow\) \textit{bitmap filtering}
Summary

• How can we speed up computation?
  – Materialized Views
  – Indexing structures
    • Bitmap index / Projection index
    • Join index
    • Bitmap-join index
    • Indexing fact and dimension tables
  – Partitioning
Indexing Fact and Dimension Tables

• Type of index depends on type of attributes

```
<table>
<thead>
<tr>
<th>Distinct values</th>
<th>Few</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Bitmap</td>
<td>Compressed bitmap Projection index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B+-tree</td>
</tr>
<tr>
<td>High</td>
<td>B+-Tree</td>
<td>B+-tree</td>
</tr>
<tr>
<td></td>
<td>Bitmap</td>
<td></td>
</tr>
</tbody>
</table>
```

• Optimal set of indices depends on workload
  – attribute often used for slicing → bitmap index
  – Key attribute → B+-tree
Indexing Dimension Tables

• Useful indices:
  – Dimensional attribute that is often used for slicing
    • Low cardinality $\rightarrow$ bitmap/bitmap-join index
    • High cardinality $\rightarrow$ B+-tree index
  – B+-tree index on surrogate key
    (necessary for joins with the fact table!)
  – Foreign keys (snowflake schema)
Indexing the Fact Table

• Index(es) on the foreign keys (for joining with dimension tables)
  – Index on attributes (A,B,C) helps queries that join on A, AB, ABC, but not on the other attribute sets
  – Often useful to have multiple indices for different orders/for foreign keys separately
Example

Customer
- CustKey
- OLTP_Custkey
- City
- Country
- Age-Cat

Sale (CustKey)
- CustKey
- DateKey
- ProdKey
- Quantity
- UnitPrice
- VATrate

Product
- ProdKey
- OLTP_Prodkey
- Brand
- Type

Date
- DateKey
- Date
- DayOfWeek
- Month
- Year

Customer (CustKey) Btree
Customer (City) Bitmap
Sale (Country) Bitmap-join
Sale (CustKey) Btree
Sale (DateKey) Btree
Sale (ProdKey) Btree

Product (ProdKey) Btree
Sale (Brand) Bitmap-join
Select Type, Month, sum(quantity)
From Customer JOIN Sale JOIN Product JOIN Date
Where
  C.City="Brussels" and
  Brand="Brand A"
Group by Type, Month
Select Type, Month, sum(quantity)
From Customer JOIN Sale JOIN Product JOIN Date
Where C.City="Brussels" and Brand="Brand A"
Group by Type, Month
Example

• Different options:
  – Use **bitmap-join index** to directly select facts about “Brand A”
  – **Aggregate by Type, Cust, Date** (reduce # tuples in join)
  – Use Btrees on primary keys in **Customer** and **Date** table to join in these tables (index nested-loop)
  – Filter tuples of join with “City=Brussels”
  – Group by Type, Month

```
Select Type, Month, sum(quantity)
From Customer JOIN Sale JOIN Product JOIN Date
Where
  C.City="Brussels" and
  Brand="Brand A"
Group by Type, Month
```
Example

• Different options:
  – Use **bitmap-join index** to directly select facts about “Brand A”
  – **Aggregate by Type, Cust, Date** (reduce # tuples in join)
  – Filter Customer on “City=Brussels” using **bitmap idx**
  – Join filtered Customer with Fact table (nested loop)
  – Use **Btree on primary keys in Date table** to join with fact table (index nested-loop)
  – Group by Type, Month

```sql
Select Type, Month, sum(quantity)
From Customer JOIN Sale JOIN Product JOIN Date
Where
  C.City="Brussels" and
  Brand="Brand A"
Group by Type, Month
```
Example

- Different options:
  - Filter Customer on “City=Brussels” using bitmap idx
  - Join Customer and fact table using Btree on foreign key CustKey in fact table (index nested-loop)
  - Use Btree on primary keys in Date and Product table to join (index nested-loop)
  - Filter on Brand=“Brand A”
  - Group by Type, Month

```
Select Type, Month, sum(quantity)
From Customer JOIN Sale JOIN Product JOIN Date
Where
  C.City=“Brussels” and
  Brand=“Brand A”
Group by Type, Month
```
Example

• Based on an estimate of the cost, optimizer will select the cheapest plan
  – Very hard to predict
  – Depends on database statistics

• Based on usage and database statistics index selection should be revised regularly
  – Remove indices that are not used
  – Add indices to speed up slow queries
Summary

• How can we speed up computation?
  – Materialized Views
  – Indexing structures
    • Bitmap index / Projection index
    • Join index
    • Bitmap-join index
    • Indexing fact and dimension tables
  – Partitioning
Partitioning

- Separate database/tables/indices over different partitions
  - Horizontal partitioning: every partition holds a subset of the *tuples*
    E.g., partition fact table by *month*
  - Vertical partitioning: every partition holds a subset of the *attributes*
Partitioning

• Advantages of horizontal partitioning
  – Easier for data warehouse refresh
    • No need to rebuild index for the whole table
  – Ease of maintenance; e.g., removing an outdated partition

• Disadvantage:
  – Overhead & reduces efficiency of indexing, especially if query spans many partitions
    → optimal partitioning depends on workload
Conclusion

• Bitmap index, join-index, bitmap-join index:
  – Speedup selection queries with arbitrary Boolean combinations of indexed attributes
    • Very interesting for ad-hoc analytical queries
  – Not easy to update
    • Not suitable for operational databases: inserts and deletes
    • Typically these indices are completely rebuild after bulk inserts

Typical for Data Warehouses; less suitable for OLTP