Data Warehousing
Indexing

Esteban Zimányi
ezimanyi@ulb.ac.be
Slides by Toon Calders

ULB
ECOLE POLYTECHNIQUE DE BRUXELLES

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 (part, supplier, customer)
  – After partial materialization:
    Answer (supplier) from (supplier, customer)

How Does it Fit In?

- Not all queries are of the type

```
SELECT D1, …, Dk, sum(M)
FROM R
GROUP BY D1, …, Dk
```

How Does it Fit In?

- Example of another type of query:

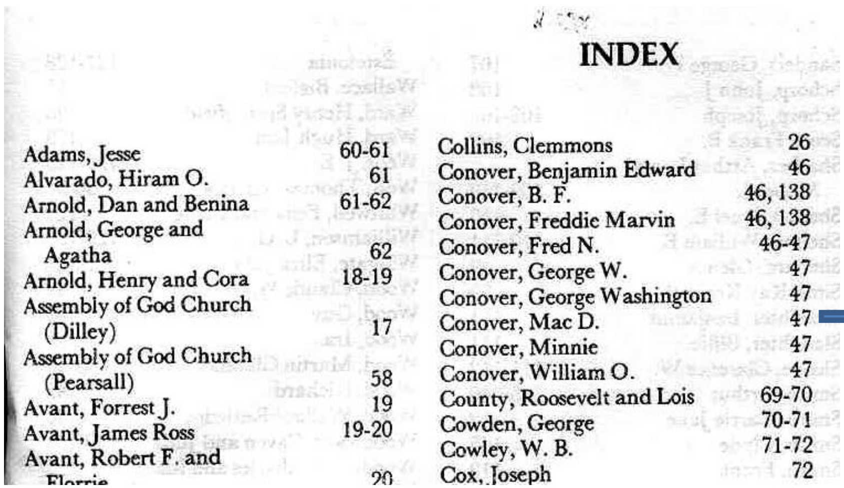
```
SELECT supplier, year, min(price)
FROM "cube"
WHERE
      product = "toilet paper"
      and (year = 2009 or year = 2010)
GROUP BY supplier, year
```

Indexing Principle

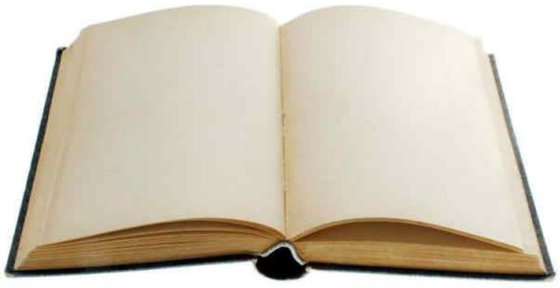
No index

Indexing Principle

No index



INDEX



Indexing Principle

- Database Equivalent

No index → Expensive

Full table scan

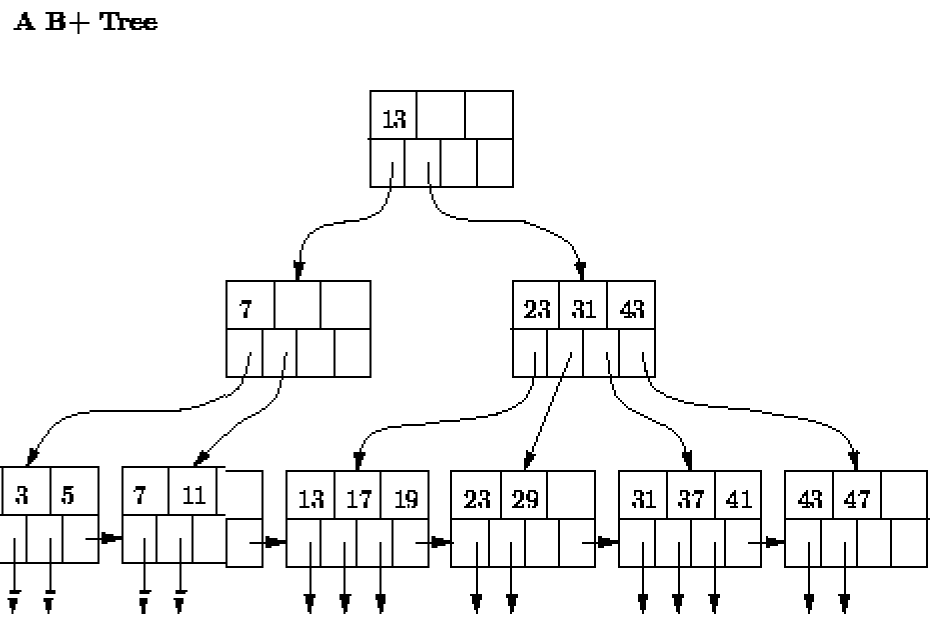
Index → Inexpensive

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

Product	Country	Sales
TV	Ireland	20
TV	France	126
HiFi	Germany	56
PC	Ireland	23
TV	France	138
PC	Germany	48

- Index for Country:

Ireland	100100
France	010010
Germany	001001

Bitmap Index: Example

Product	Country	Sales
TV	Ireland	20
TV	France	126
HiFi	Germany	56
PC	Ireland	23
TV	France	138
PC	Germany	48

- Index for Country:

Ireland	100100
France	010010
Germany	001001

Index for Product

TV	110010
HiFi	001000
PC	000101

Bitmap Index: Example

- 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 kati ota 1 kati ota 0

  – Reduces storage requirements significantly

  – Logical operations can work directly on RLE

- BTree maps values to the (variable length) bitmaps

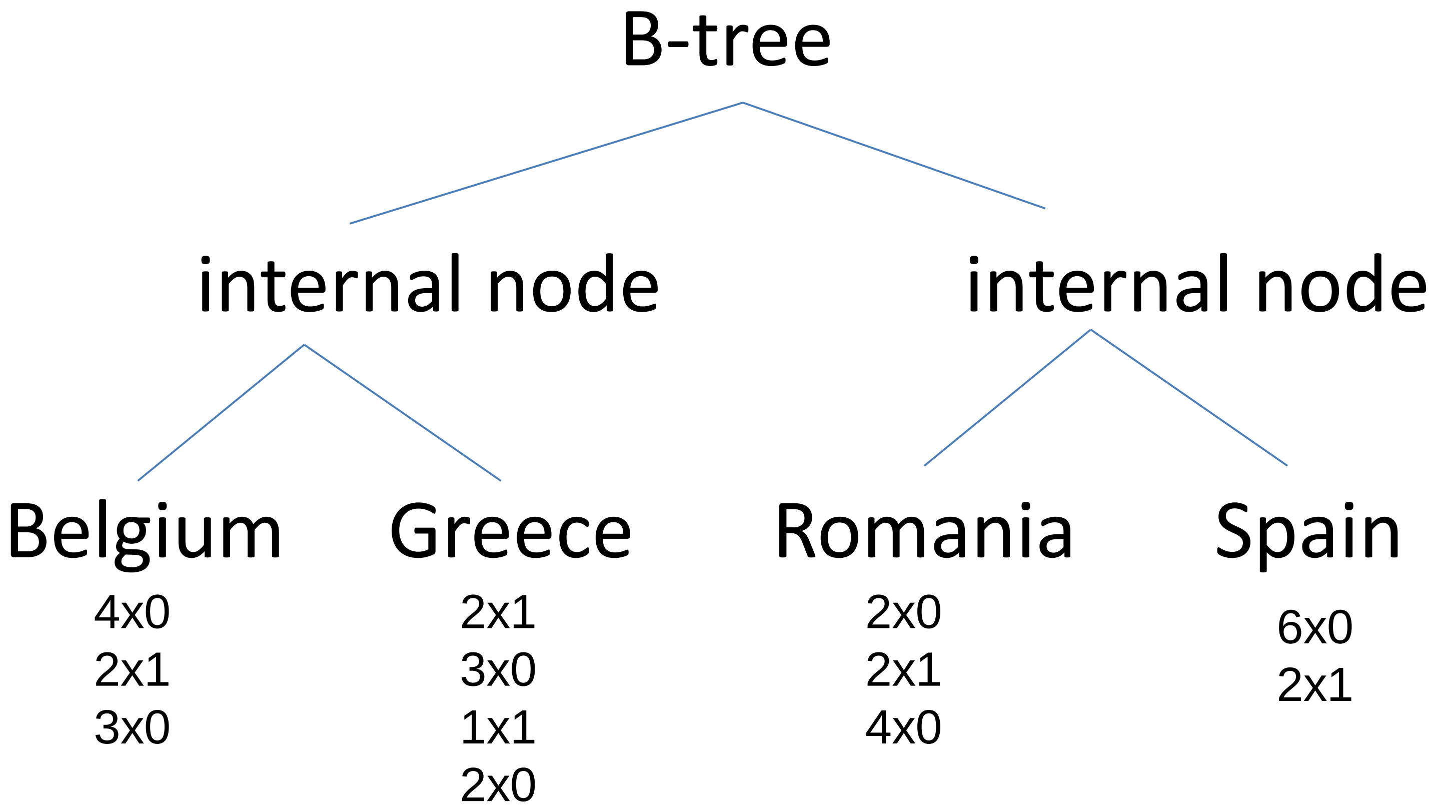
Working Directly on RLE

- Brussels: 00000001100001000000
  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

bitmap along with btree



Leafs contain the bitmaps

Variants of Bitmap Index

Disjoint ranges
use case - age

Overlapping ranges
use case: for temperature
because we might have queries most frequently asked as get all temp>10

which one is better depende on type of query we will be doing

- 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

    temperature > 5 → b(]0,5]) | b(]5,10]) | …
- Other option: make overlapping bitmaps
  - Bitmap for all tuples with t>-10, t>-5, t>0, t>5, …

    temperature between 0 and 5: b(t>0) & - 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, …

indexing string values

Customer	City
C1	Antwerp
C2	Brussels
C3	Eindhoven
C4	Eindhoven

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, …

string represented in the form of integer
and somewhere else will be having mapping

Customer	City
C1	0000
C2	0001
C3	1011
C4	1011

Bit 1	Bit 2	Bit 3	Bit 4
0	0	0	0
0	0	0	1
1	0	1	1
1	0	1	1

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, …

Customer	City
C1	0000
C2	0001
C3	1011
C4	1011

Bit 1	Bit 2	Bit 3	Bit 4
0	0	0	0
0	0	0	1
1	0	1	1
1	0	1	1

Antwerp or Brussels → (!b1 & !b2 &!b3 & !b4) | (!b1 & !b2 &!b3 & b4)
→ (!b1 & !b2 &!b3)

Bitmap Index: Pros and Cons

if the value has updated find that row, and update value accordingly, which is hard due to compression
more useful for read only cases

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

- Particularly useful when there are multiple bitmap indices

not useful for price
if we need to do, can create ranges and apply bin concept

- *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

between two refresh these values are kept in same order

Customer	Age	City
C1	30	Antwerp
C2	34	Brussels
C3	39	Eindhoven
C4	65	Eindhoven

→

Customer
C1
C2
C3
C4

City
Antwerp
Brussels
Eindhoven
Eindhoven

Age
30
34
39
65

Added to the database

(Vertical) Projection Index

- Advantage:
  - better data locality if only few attributes are needed

    needed info are brouught from disk to memory
  - 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 → rows/RIDs in R
- Join indices:
  - Value of R.A → RIDs in S
- Data warehouse:
  - Attribute in dimension table → rows in fact table
  - Join indexes can span multiple dimensions

Join Index: Example

Sales

Date	pID	Client
10/5/12	1	Jack
10/5/12	1	Pete
13/5/12	3	John
14/5/12	2	Mary

$\bowtie$

Products

pID	pName	Category	Price
1	Jacket	Non-food	10
2	Bread	Food	2.1
3	Beer	Food	1.5
4	Paper	Non-food	1.2

$\Downarrow$

Date	pID	Client	pName	Category	Price
10/5/12	1	Jack	Jacket	Non-food	10
10/5/12	1	Pete	Jacket	Non-food	10
13/5/12	3	John	Beer	Food	1.5
14/5/12	2	Mary	Bread	Food	2.1

Join Index: Example

Sales

Date	pID	Client
10/5/12	1	Jack
10/5/12	1	Pete
13/5/12	3	John
14/5/12	2	Mary

Products

pID	pName	Category	Price
1	Jacket	Non-food	10
2	Bread	Food	2.1
3	Beer	Food	1.5
4	Paper	Non-food	1.2

$\bowtie$

SP_category_jidx

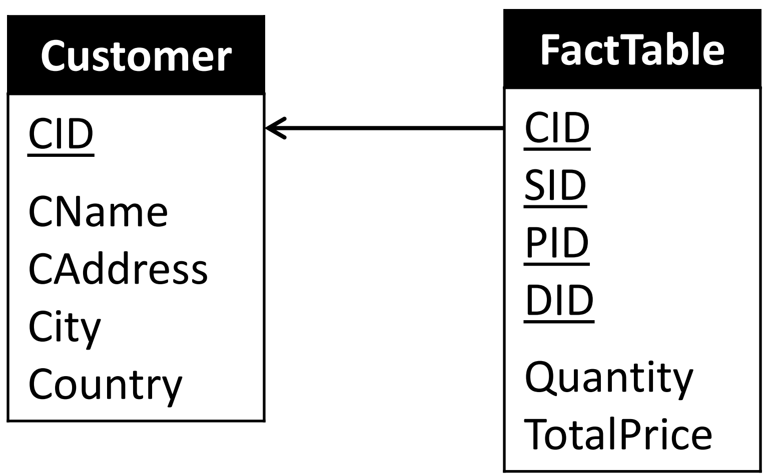
Category	RID
Non-food	r1,r2
Food	r3,r4

cat materialized

Date	pID	Client	pName	Category	Price
10/5/12	1	Jack	Jacket	Non-food	10
10/5/12	1	Pete	Jacket	Non-food	10
13/5/12	3	John	Beer	Food	1.5
14/5/12	2	Mary	Bread	Food	2.1

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

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.

instead of repeating non food two time
we can add lookup

Products

pID	pName	Category	Price	RIDSales
1	Jacket	Non-food	10	r1, r2
2	Bread	Food	2.1	r4
3	Beer	Food	1.5	r3
4	Paper	Non-food	1.2	

Sales

Date	pID	Client
10/5/12	1	Jack
10/5/12	1	Pete
13/5/12	3	John
14/5/12	2	Mary

Bitmap-join index

- Logical combination of bitmap index and join index

Sales

Date	pID	Client
10/5/12	1	Jack
10/5/12	1	Pete
13/5/12	3	John
14/5/12	2	Mary

SP_category_jidx

Category	RID
Non-food	r1,r2
Food	r3,r4

SP_category_**bj**idx

Category	Bitmap
Non-food	**1100**
Food	**0011**

Bitmap-Join Index: Example

Date	pID	Client
10/5/12	1	Jack
10/5/12	1	Pete
13/5/12	3	John
14/5/12	2	Mary

SP_category_bjidx

Category	Bitmap
Non-food	1100
Food	0011

SC_city_bjidx

City	Bitmap
Brussels	1001
Eindhoven	0110

SELECT date

FROM Sales S JOIN Product P JOIN Customer C

WHERE

P.Category = "Food" and

C.City = "Brussels";

Bitmap-Join Index: Example

Date	pID	Client
10/5/12	1	Jack
10/5/12	1	Pete
13/5/12	3	John
14/5/12	**2**	**Mary**

SP_category_bjidx

Category	Bitmap
Non-food	1100
Food	**0011**

SC_city_bjidx

City	Bitmap
Brussels	**1001**
Eindhoven	0110

SELECT date

FROM Sales S JOIN Product P JOIN Customer C

WHERE

P.Category = “Food” and

C.City = “Brussels”;

0011 & 1001 → 0001

Indices in Practice

- Several commercial products implement bitmap, join & bitmap-join index
  - E.g., Oracle (EE) offers a compressed bitmap index

    CREATE BITMAP INDEX cust_sales_bji
    ON sales(customers.city)
    FROM sales, customers
    WHERE sales.client = customers.name;

- Some products build bitmaps *on the fly*
  - E.g., SQLServer 2008 → *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

higher the age criteria less chance of getting selected in university records

Distinct values

Query selectivity

	Few	Many
Low	**Bitmap**	Compressed bitmap Projection index B+-tree
High	B+-Tree Bitmap	**B+-tree**

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

Indexing Dimension Tables

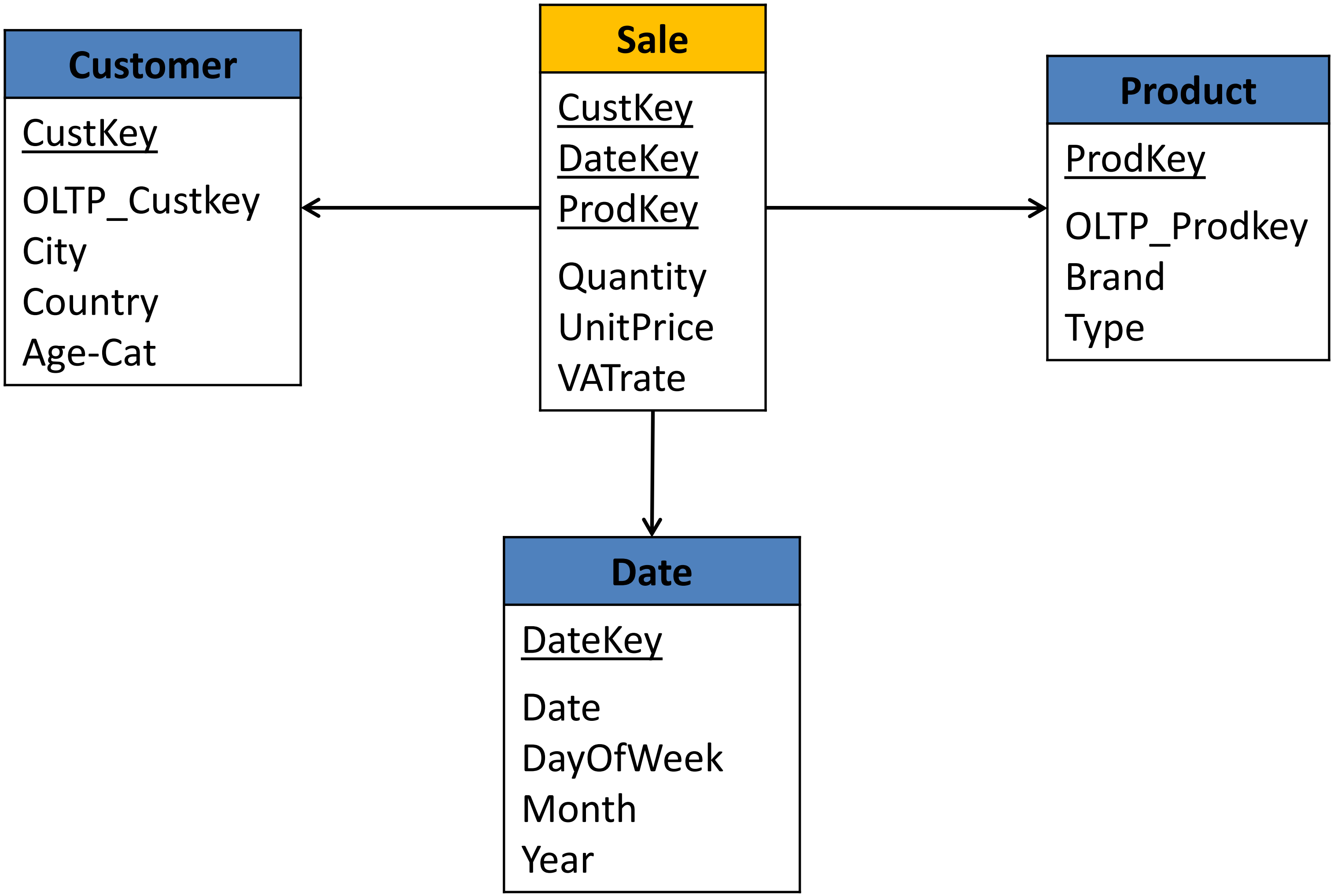
- Useful indices:
  - Dimensional attribute that is often used for slicing
    - Low cardinality → bitmap/bitmap-join index
    - High cardinality → B+-tree index
  - B+-tree index on surrogate key
    (necessary for joins with the fact table!)
  - Foreign keys (snowflake schema)

**
Ex: Client City State Country
Snowflake schema - B-tree Index or bitmap join index
Star schema - bitmap index or bitmap join index

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

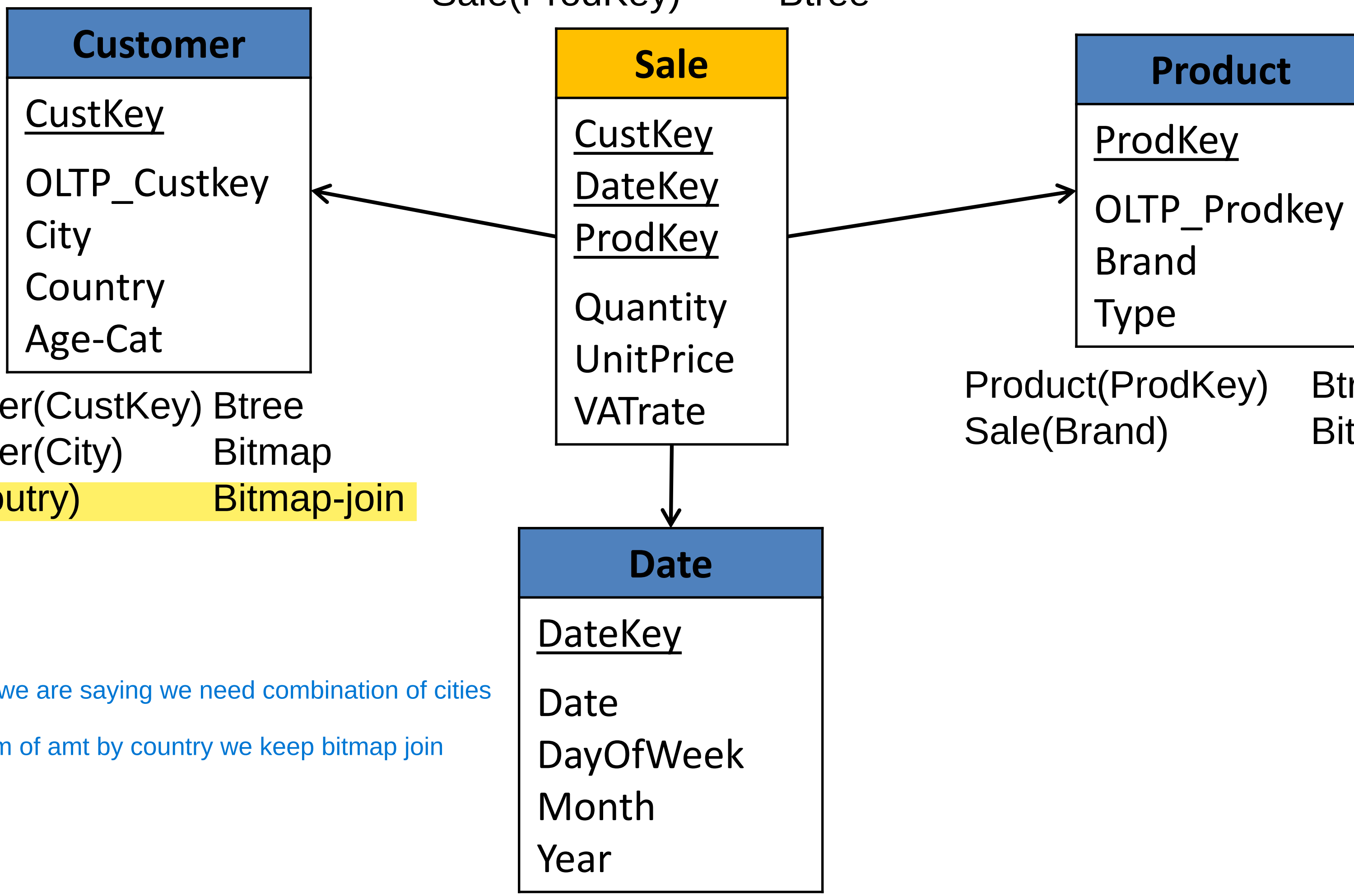
Example


Customer
CustKey
OLTP_Custkey
City
Country
Age-Cat
Sale
CustKey
DateKey
ProdKey
Quantity
UnitPrice
VATrate
Product
ProdKey
OLTP_Prodkey
Brand
Type
Date
DateKey
Date
DayOfWeek
Month
Year

Example

we don't keep index for product name because we already have prodkey and have high variance as well

Sale(CustKey) Btree
Sale(DateKey) Btree
Sale(ProdKey) Btree



Customer(CustKey)	Btree
Customer(City)	Bitmap
Sale(Coutry)	Bitmap-join

Product(ProdKey)	Btree
Sale(Brand)	Bitmap-join

bitmap halda we are saying we need combination of cities

if we want sum of amt by country we keep bitmap join

Date(DateKey) Btree

Example

Sale(CustKey) Btree
Sale(DateKey) Btree
Sale(ProdKey) Btree

Customer

CustKey

OLTP_Custkey
City
Country
Age-Cat

Customer(CustKey) Btree
Customer(City) Bitmap
Sale(Coutry) Bitmap-join

Sale

CustKey
DateKey
ProdKey

Quantity
UnitPrice
VATrate

Product

ProdKey

OLTP_Prodkey
Brand
Type

Product(ProdKey) Btree
Sale(Brand) Bitmap-join

Date

DateKey

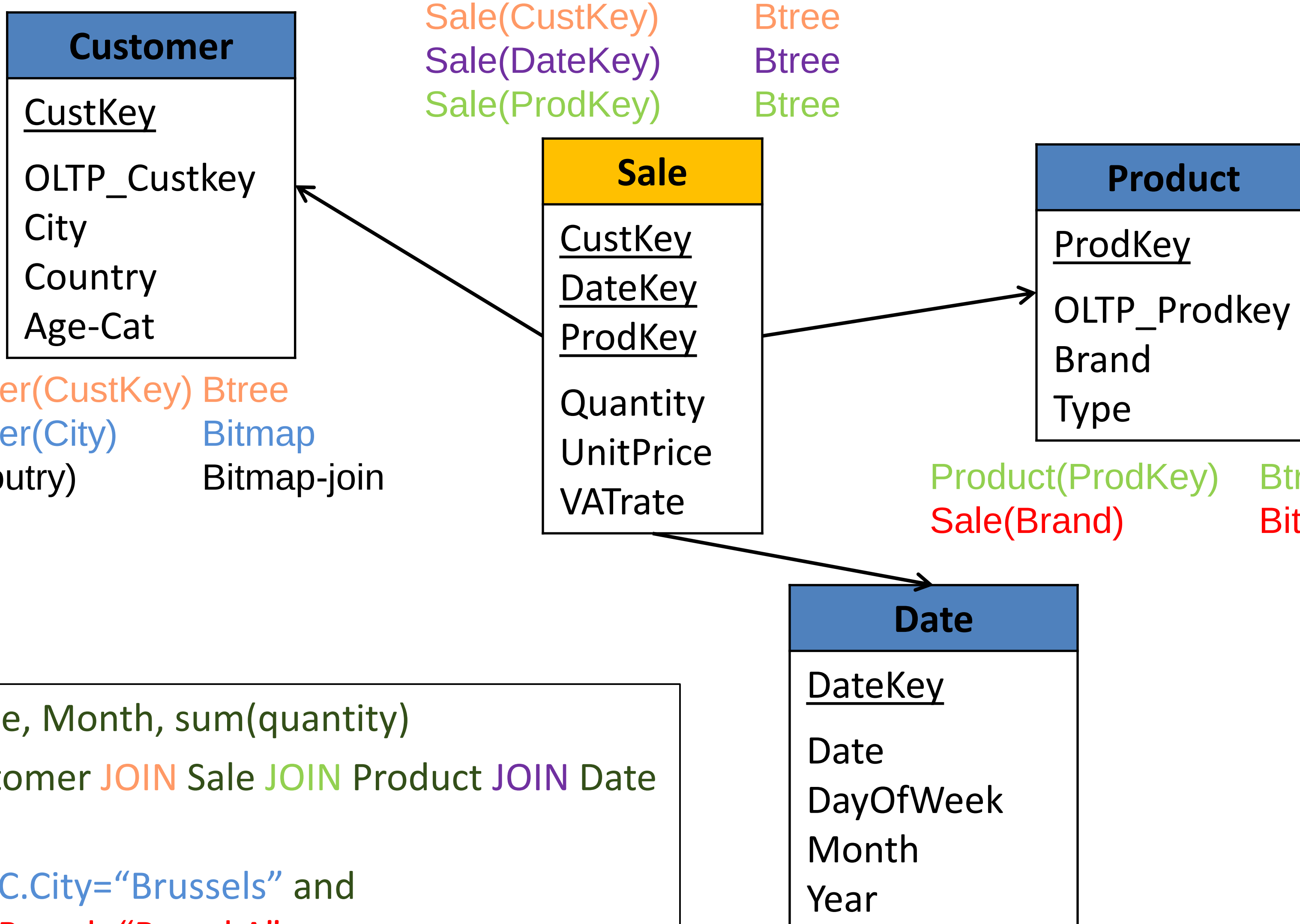
Date
DayOfWeek
Month
Year

Date(DateKey) Btree

```
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

Sale(CustKey) Btree
Sale(DateKey) Btree
Sale(ProdKey) Btree



Customer(CustKey) Btree
Customer(City) Bitmap
Sale(Coutry) Bitmap-join

Product(ProdKey) Btree
Sale(Brand) Bitmap-join

Date(DateKey) Btree

```
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

```
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