

Multidimensional Databases

Logical Design
Peter Scheuermann

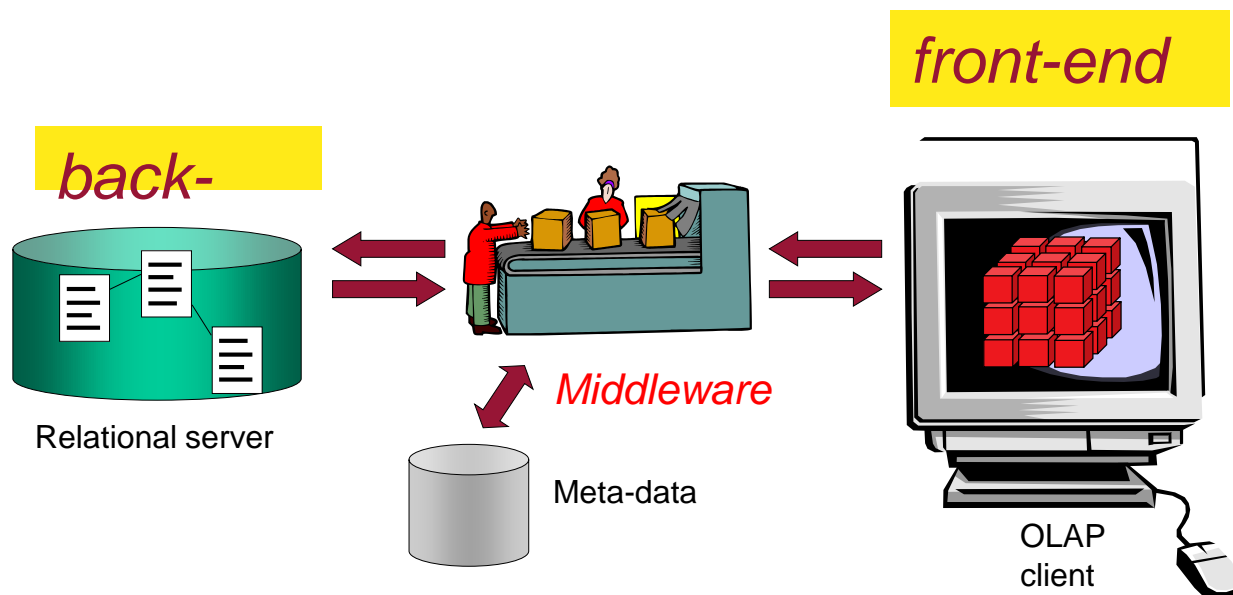
ROLAP

- Relational OLAP
- Data stored in relational tables
 - Star (or snowflake) schemas used for modeling
 - SQL used for querying
- Pros
 - Leverages investments in relational technology
 - Scalable (billions of facts)
 - Flexible, designs easier to change
 - New, performance enhancing techniques adapted from MOLAP
 - ◆ Indices, materialized views
- Cons
 - Storage use (often 3-4 times MOLAP)
 - Response times

Product ID	Store ID	Sales
1	3	2
2	1	7
3	2	3
...

ROLAP Architecture

Some open source systems: Mondrian
Pentaho



MOLAP

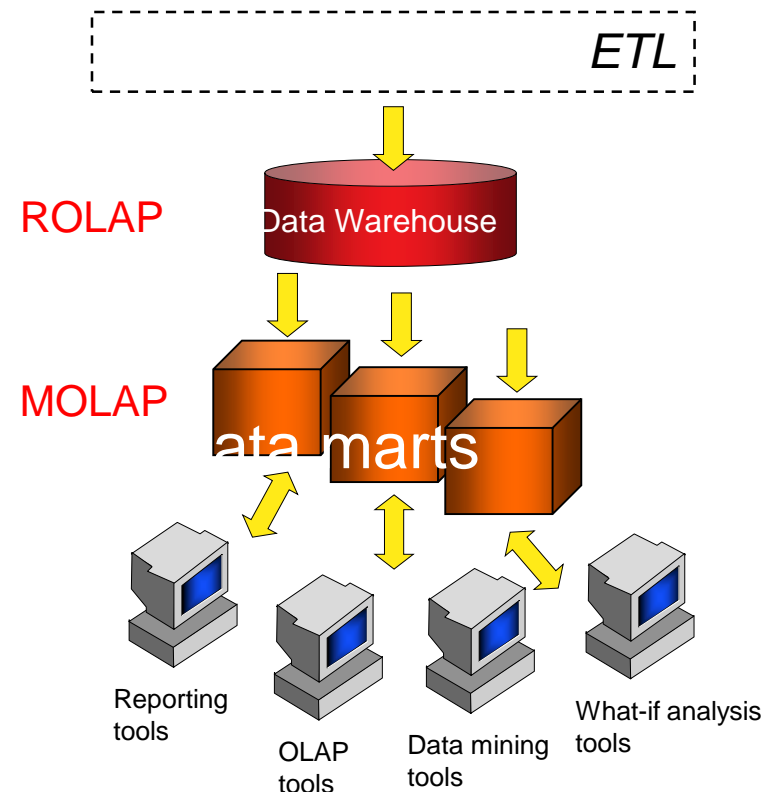
- Multidimensional OLAP
- Data stored in special multidimensional data structures
 - E.g., multidimensional array on hard disk
- Pros
 - Less storage use (“foreign keys” not stored)
 - Faster query response times
- Cons
 - Up till now scalability not so great
 - Less flexible, e.g., cube must be re-computed when design changes
 - Not as open technology
 - Lack of standard logical model
- Some commercial systems: Microsoft Analysis Services

MOLAP data cube

$d_2 \setminus d_1$	1	2	3
1	0	7	0
2	0	0	3
3	2	0	0

HOLAP (Hybrid OLAP)

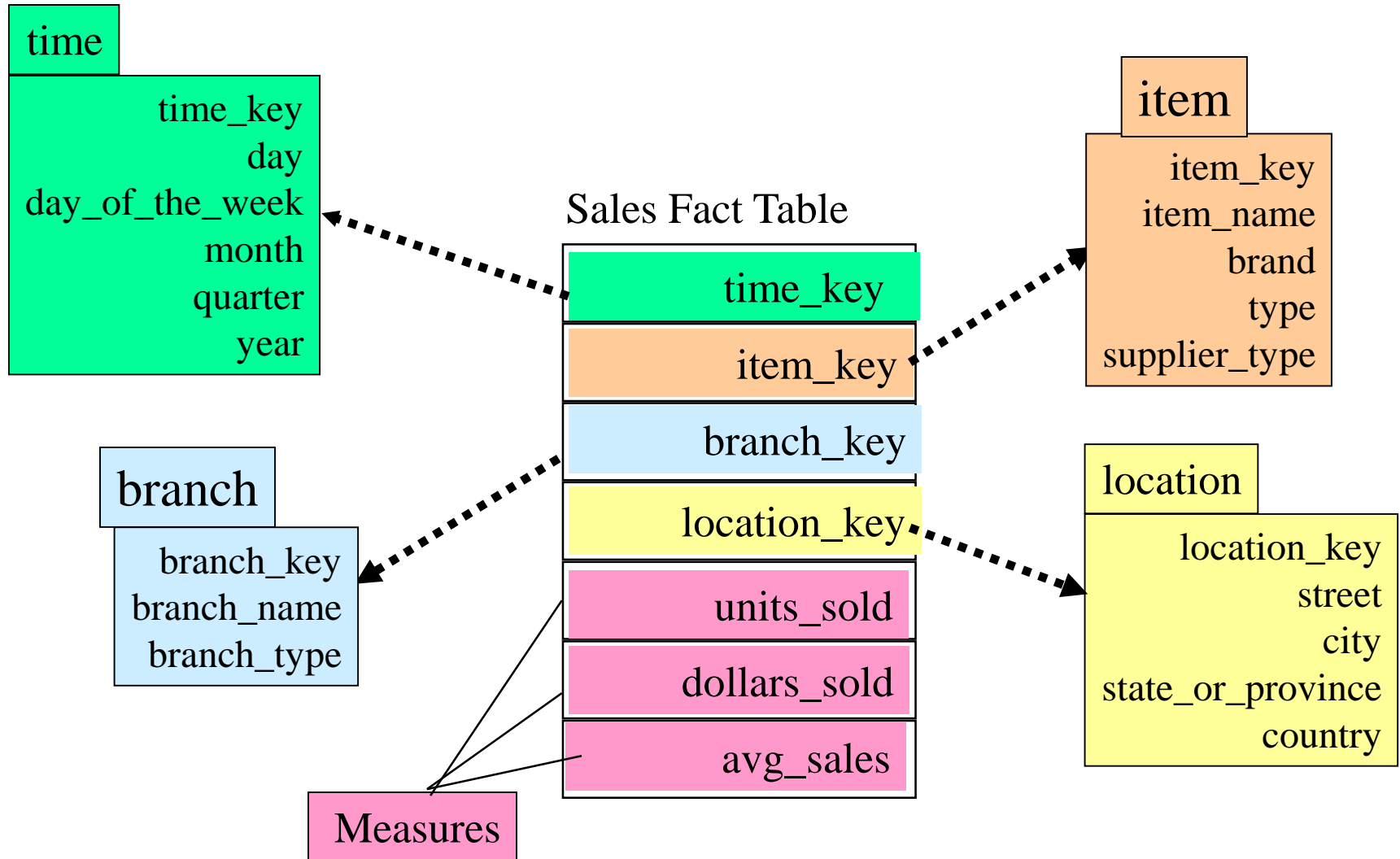
- **ROLAP and MOLAP elements are combined into a single architecture**
 - the largest amount of data is stored in an RDBMS to avoid the problems caused by sparseness,
 - **MOLAP stores only the (aggregated) information users most frequently need to access**
 - Dense cubes are stored in MOLAP, sparse cubes in a ROLAP
 - Some commercial systems: MicroStrategy, Business Objects



The Star Schema

- Multidimensional modeling in relational systems is based on the *star schema* and on its variants
- A star schema consists of the following:
 - A set of *dimension tables* (DT_1, \dots, DT_n), each corresponding to a dimension. Every DT_i has a primary (typically surrogate) key (k_i) and a set of attributes at different aggregation levels
 - A *fact table* (FT) including *measures*. An *FT primary key* is the composition of the set of foreign keys (k_1 through k_n) referencing dimension tables
- Dimension tables are not in 3rd normal form because transitive functional dependencies exist due to the presence of all the attributes of a hierarchy in the same relation
 - There is some redundancy
 - The number of joins needed to retrieve information is reduced

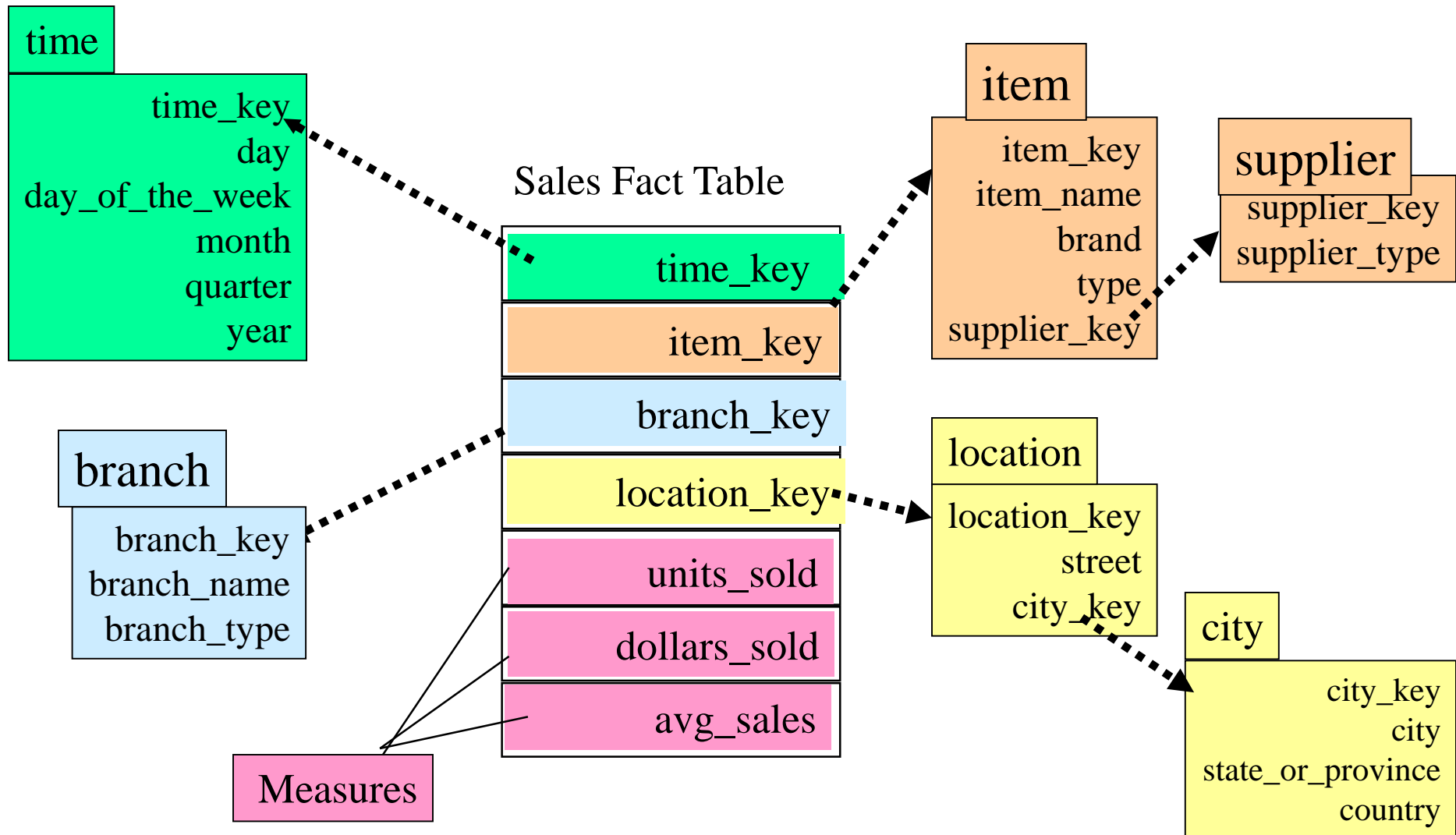
Example of Star Schema



Snowflake Schema

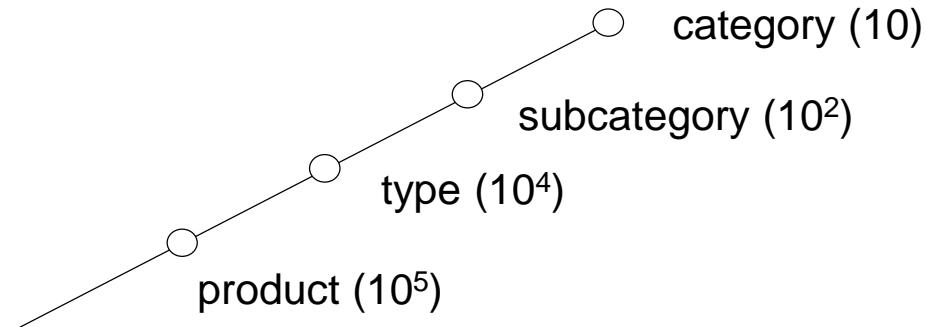
- A refinement of the star schema where some dimensional hierarchy is normalized into a set of smaller dimension tables, forming a shape similar to snowflake
- To break down a schema effectively, you need for all those attributes that—directly or transitively—depend on the snowflaking attribute (that is, on the natural key of the new relation) to be part of the new relation

Example of Snowflake Schema



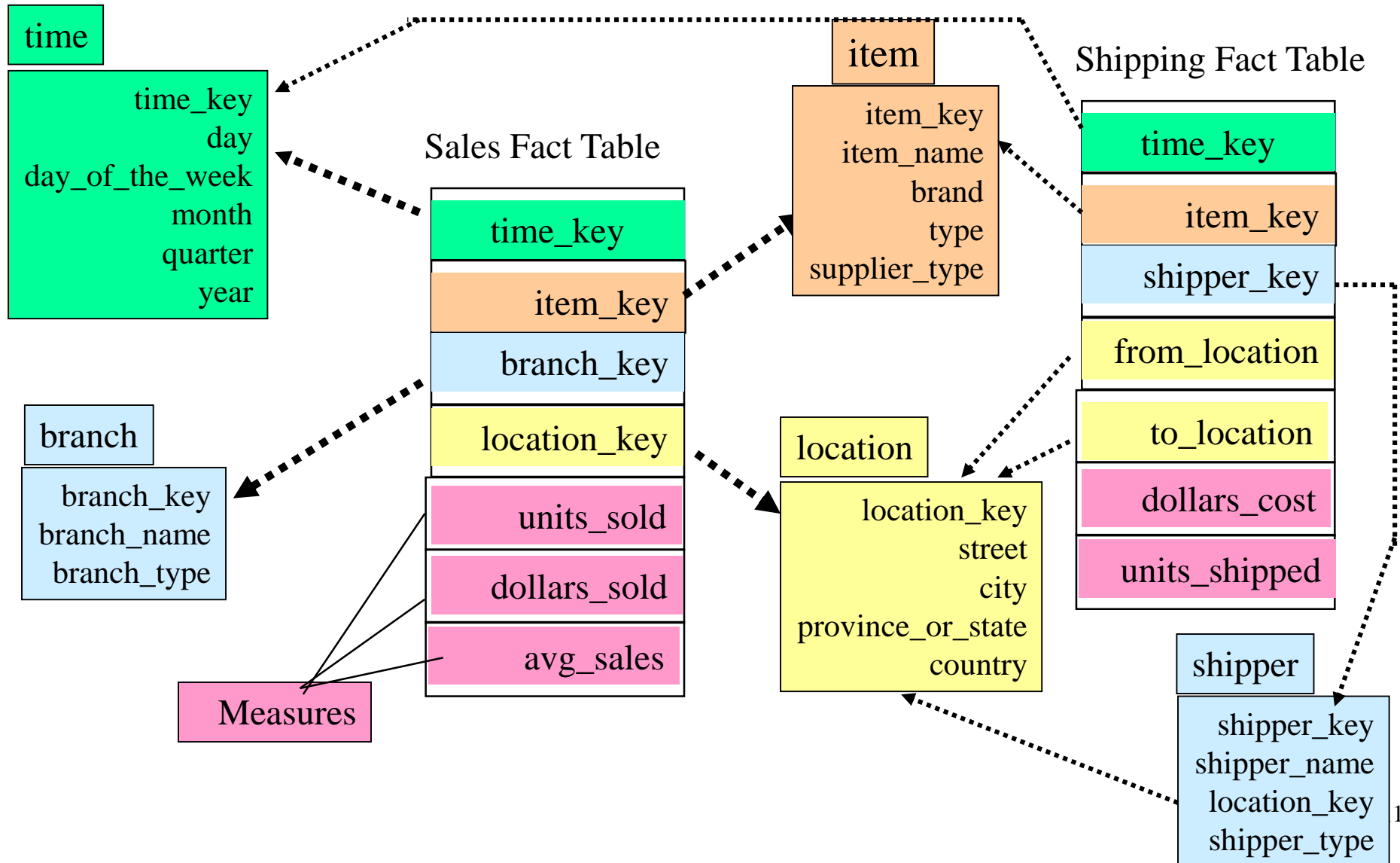
Star vs. snowflake

- Snowflaking may be useful when:
 - The ratio between the cardinalities of the primary and secondary DTs is high, because in this case it leads to a relevant space savings



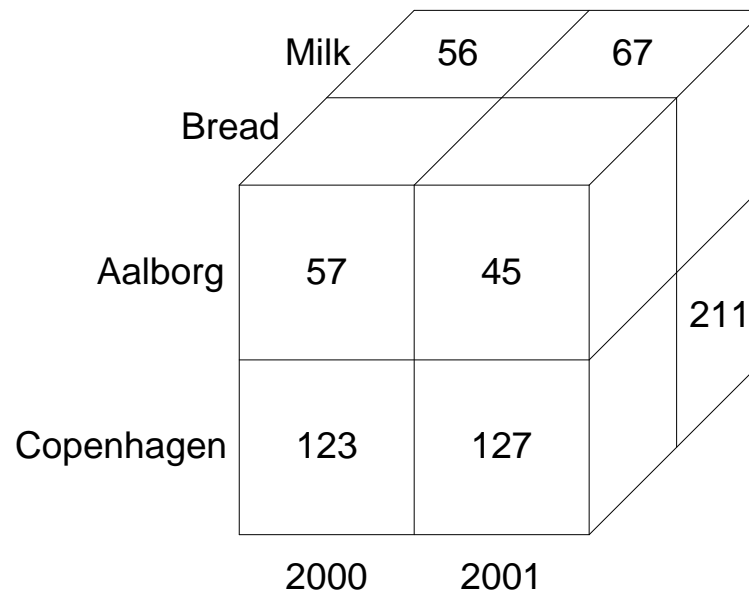
Fact Constellation

Multiple fact tables share dimension table

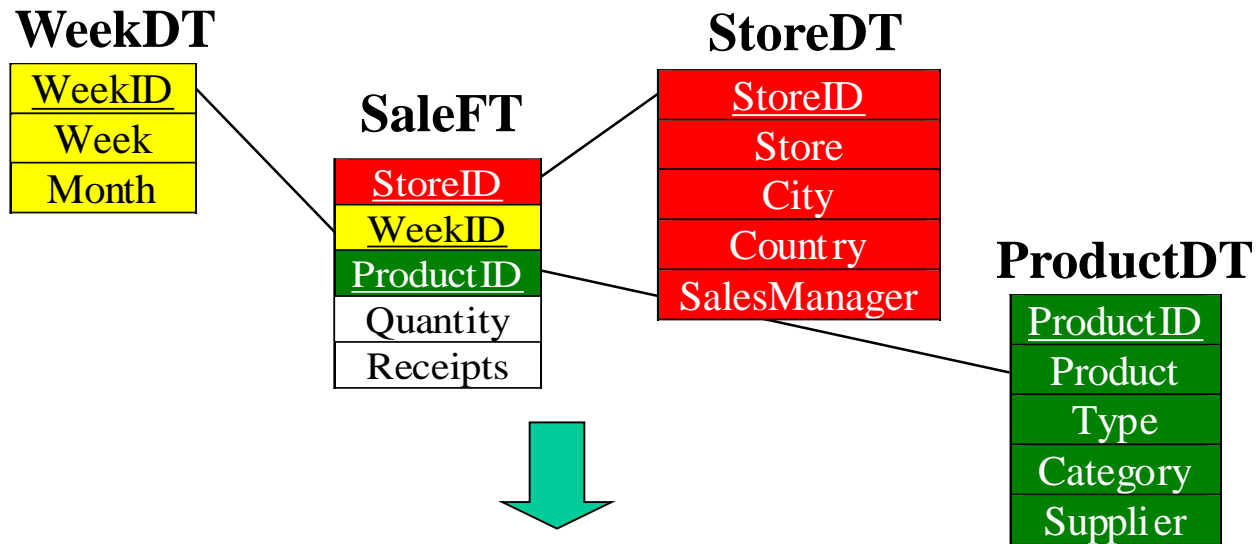


(Relational) OLAP Queries

- **Two** kinds of queries
 - **Navigation queries** examine one dimension
 - ♦ `SELECT DISTINCT I FROM d [WHERE p] // p is predicate`
 - **Aggregation queries** summarize fact data
 - ♦ `SELECT d1.I1, d2.I2, SUM(f.m) FROM d1, d2, f
WHERE f.dk1 = d1.dk1 AND f.dk2 = d2.dk2 [AND p...]
GROUP BY d1.I1, d2.I2`
- Fast, interactive analysis of large amounts of data



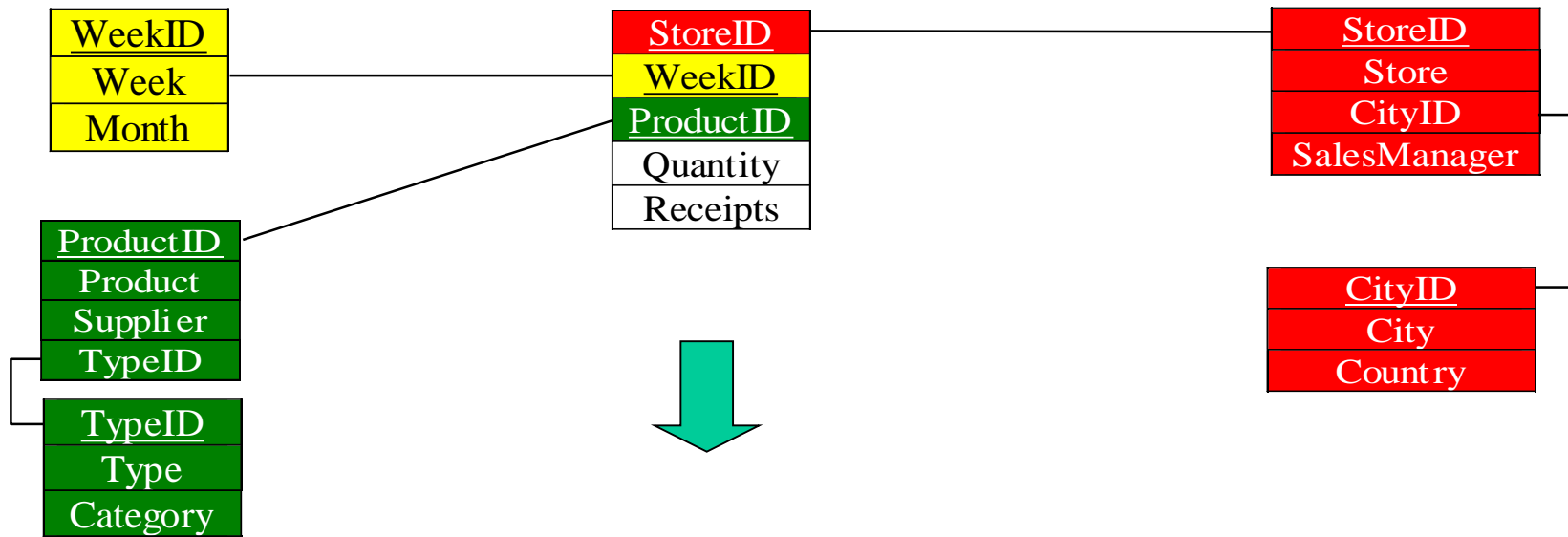
OLAP queries on a star schema



Total quantity sold for each product type, week, and city, only for food products

```
SELECT    City, Week, Type, SUM(Quantity)
FROM      WeekDT, StoreDT, ProductDT, SaleFT
WHERE     WeekDT.WeekID = SaleFT.WeekID AND
          StoreDT.StoreID = SaleFT.StoreID AND
          ProductDT.ProductID = SaleFT.ProductID
          AND ProductDT.Category = 'Food'
GROUP BY  City, Week, Type;
```

OLAP queries on a snowflake schema



Total quantity sold for each product type, week, and city, only for food products

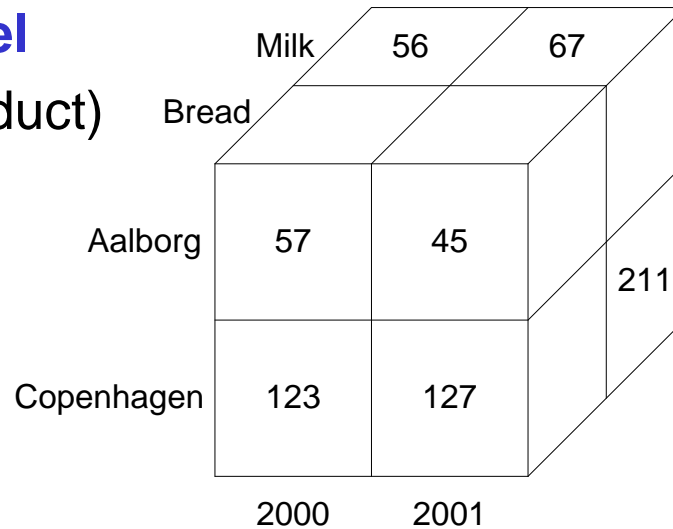
```
SELECT  City, Week, Type, SUM(Quantity)
FROM    WeekDT, StoreDT, ProductDT, CityDT, TypeDT,
SaleFT
WHERE   WeekDT.WeekID = SaleFT.WeekID AND
        StoreDT.StoreID = SaleFT.StoreID AND
        ProductDT.ProductID = SaleFT.ProductID AND
        StoreDT.CityID = CityDT.CityID AND
        ProductDT.TypeID = TypeDT.TypeID AND
        ProductDT.Category = 'Food'
GROUP BY City, Week, Type;
```

Typical OLAP Operations

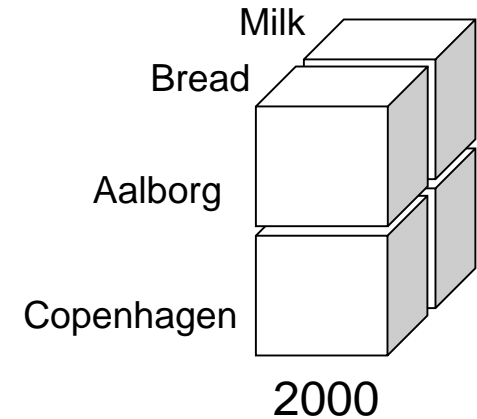
- **Roll up (drill-up): summarize data**
 - by climbing up hierarchy or by dimension reduction
- **Drill down (roll down):** reverse of roll-up
 - from higher level summary to lower level summary or detailed data, or introducing new dimensions
- **Slice and dice:** project and select
- **Pivot (rotate):**
 - reorient the cube, visualization, 3D to series of 2D planes
- Other operations
 - *drill across:* involving (across) more than one fact table
 - *drill through:* through the bottom level of the cube to its back-end relational tables (using SQL)

Typical OLAP Queries

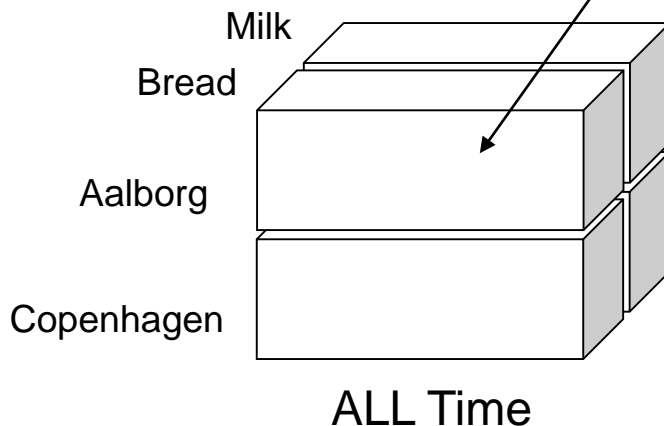
Starting level
(City, Year, Product)



Slice/Dice:

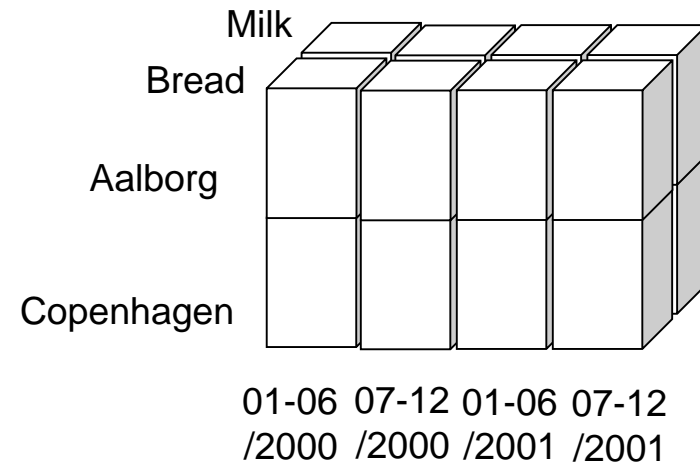


Roll-up: get overview



What is this value?

Drill-down: more detail



Temporal scenarios—Dynamic hierarchies

- The multidimensional model assumes that the events instantiating facts are **dynamic** and that the attribute values populating hierarchies are **static**
- This is not generally realistic because changes are progressively applied to hierarchies in real-world cases (**dynamic hierarchies** or **slowly-changing dimensions**)
- Using dynamic hierarchies leads to a space and performance overhead

Temporal scenarios

- **Today-for-yesterday (*up-to-date*)**
 - The events are analyzed according to the hierarchies' current configuration
 - It can be implemented on a star schema (Type I)
- **Today-or-yesterday (*historical truth*)**
 - Each event is analyzed according to the configuration the hierarchies had at the time when the event occurred
 - It can be implemented on a star schema (Type II)
- **Yesterday-for-today (*rollback*)**
 - All of the events are analyzed according to the configuration the hierarchies had at a previous time
 - Data must be logged (Type III)

Dynamic hierarchies: Type I

- They support only the *today-for-yesterday* scenario
- All the events including past ones are always interpreted from the viewpoint of the current hierarchy instance, without tracking previous instances
 - To manage this type of scenario, we need a **regular star schema**
 - As soon as a change is applied to a tuple value of the dimension table, a new value simply overwrites the old one, so all the fact table data are associated with the new dimension table value

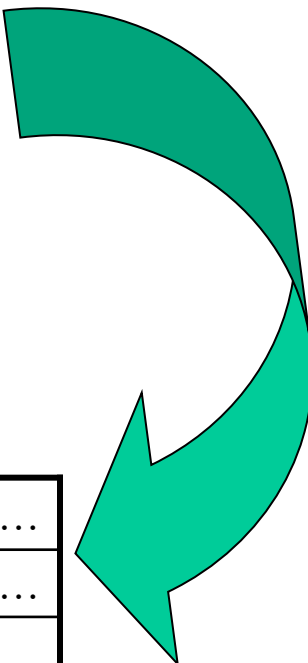
Dynamic hierarchies: Type I

Situation at 1/1/2015

<u>StoreID</u>	Store	StoreManager	...
1	EverMore	Smith	...
2	ProFitsOnly	White	...
3	SmartMart	White	...

Situation at 1/7/2015

<u>StoreID</u>	Store	StoreManager	...
1	EverMore	Smith	...
2	ProFitsOnly	White	...
3	SmartMart	Smith	...



All sales of SmartMart
are attributed to Smith
though they were
made during White's
managing

Dynamic hierarchies: Type II

- This design solution supports the *yesterday-or-today* scenario, and it can record what really happened in the past
- An event stored into a fact table has to be associated with the hierarchy instance that was valid when that event took place
 - To manage this type of scenario, you can use a regular star schema
 - Wherever a change is applied to a hierarchy, a new tuple is added to store new values into the appropriate dimension table

Dynamic hierarchies: Type II

Situation at 1/1/2015

<u>StoreID</u>	Store	StoreManager	...
1	EverMore	Smith	...
2	ProFitsOnly	White	...
3	SmartMart	White	...

Situation at 1/7/2015

<u>StoreID</u>	Store	StoreManager	...
1	EverMore	Smith	...
2	ProFitsOnly	White	...
3	SmartMart	White	...
4	SmartMart	Smith	...

After 1/7, the fact table tuples for SmartMart should always reference StoreID = 4

All the events are selected without any distinction if query constrains only attributes whose values show no previous change!

Dynamic hierarchies: Type III

- They are based on *full data logging of dimension tables* to support every temporal analysis scenario
 - **Required features:**
 - ◆ A couple of timestamps specifying the validity interval of tuples
 - ◆ A `master` attribute that specifies the key value of the original tuple from which each tuple stems
 - You should add a new tuple to the dimension table whenever an attribute changes, as Type II dynamic hierarchies do
 - The values of timestamps and `master` attributes should then be updated accordingly

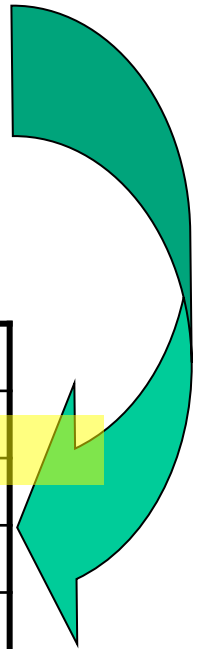
Dynamic hierarchies: Type III

Situation at 1/1/2014

<u>StoreID</u>	Store	StoreManager	...	From	To	Master
1	EverMore	Smith	...	1/1/2014	—	1
2	ProFitsOnly	White	...	1/1/2014	—	2
3	SmartMart	White	...	1/1/2014	—	3

Situation at 1/7/2015

<u>StoreID</u>	Store	StoreManager	...	From	To	Master
1	EverMore	Smith	...	1/1/2014	31/12/2014	1
2	ProFitsOnly	White	...	1/1/2014	—	2
3	SmartMart	White	...	1/1/2014	30/6/2014	3
4	SmartMart	Smith	...	1/7/2014	31/10/2014	3
5	CropShop	Smith	...	1/11/2014	—	5
6	CoreStore	Smith	...	1/1/2012	—	3
7	EverMore	White	...	1/1/2012	—	1



Dynamic hierarchies: Type III

- If you use the schema previously described, the following temporal scenarios can easily be supported:
 - **Today-for-yesterday**: The dimension table tuples that are currently valid are retrieved. Then the **master** attribute is used to retrieve all the other tuples from which each tuple is derived
 - **Yesterday-for-today**: After the user has set a specific date, the dimension table tuples that were valid at that time are found. Then the same steps as the preceding scenario are followed
 - **Today-or-yesterday**: This scenario is feasible without using timestamps because dimension table tuples are managed as in Type II dynamic hierarchies

Dynamic hierarchies: Type III

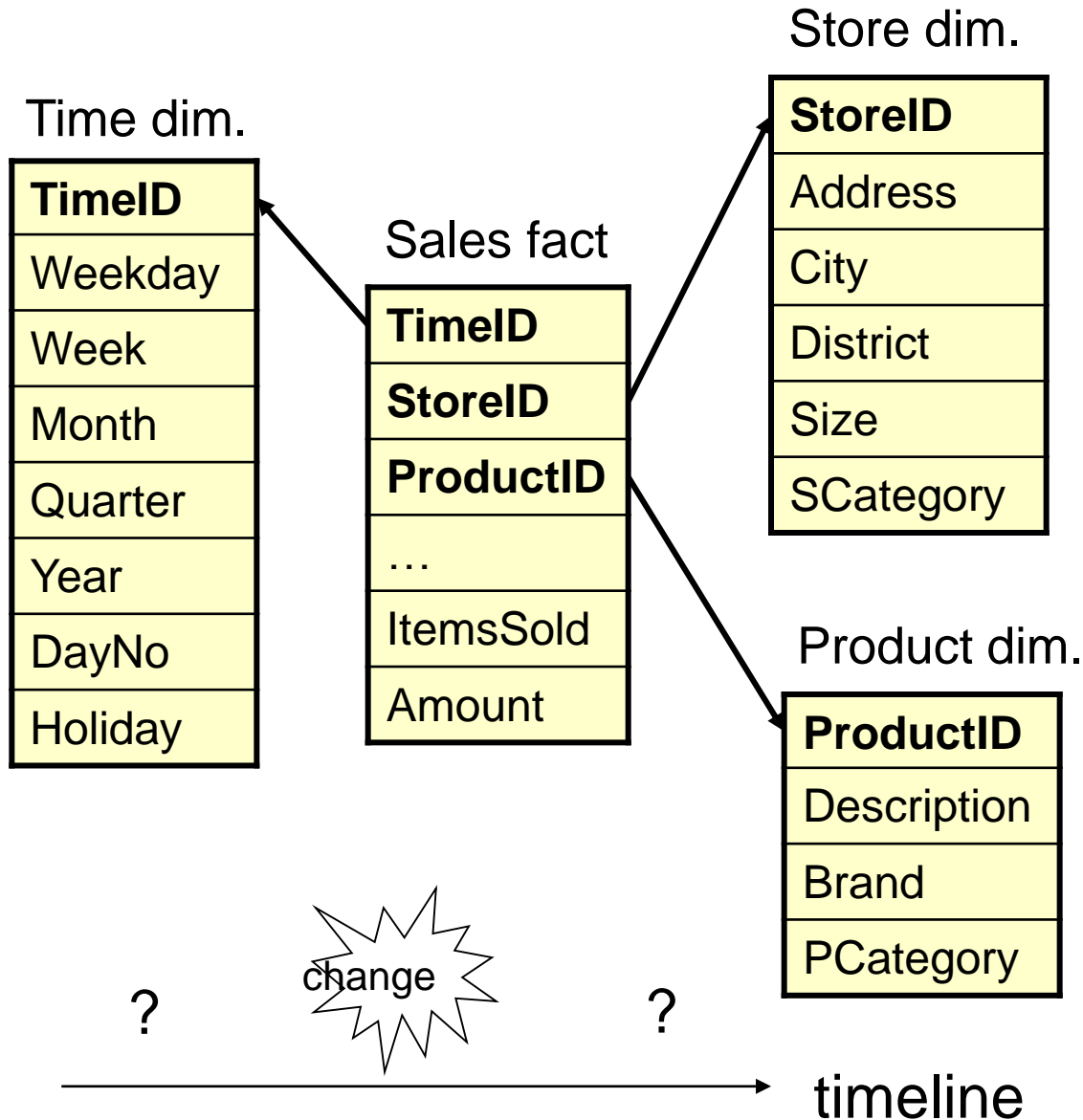
- According to the yesterday-for-today scenario, the SQL query asking for the “total amount of items sold by every sales manager if you take into account their appointment on 1/10/2014” is

```
SELECT      S1.StoreManager, SUM(Quantity)
FROM        StoreDT S1, StoreDT S2, SaleFT
WHERE       S1.From <= 1/10/2014
            AND S1.To > 1/10/2014
            AND S1.Master=S2.Master
            AND SaleFT.StoreID=S2.StoreID
GROUP BY    S1.StoreManager;
```

Handling Changes in Dimensions

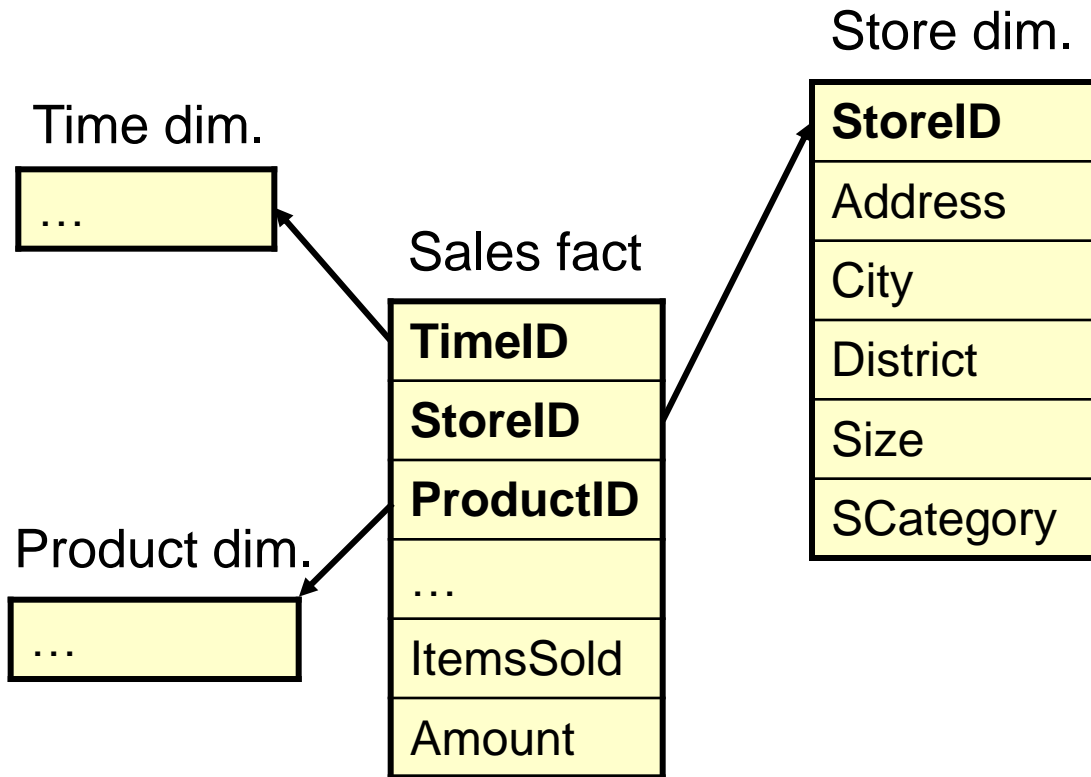
- Handling change over time
- Changes in dimensions
 - 1. No special handling
 - 2. Versioning dimension values
 - ◆ 2A. Special facts
 - ◆ 2B. Timestamping
 - 3. Capturing the previous and the current value
 - 4. Split into changing and constant attributes

Example



- Attribute values in dimensions vary over time
 - A store changes Size
 - A product changes Description
 - Districts are changed
- Problems
 - Dimensions not updated
→ DW is not up-to-date
 - Dimensions updated in a straightforward way
→ incorrect information in historical data

Example (2)



The store in Aalborg has the size of 250 sq. metres.

On a certain day, customers bought 2000 apples from that store.

Sales fact table

StoreID	...	ItemsSold	...
001		2000	

Store dimension table

StoreID	...	Size	...
001		250	

Solution 1: No Special Handling

Sales fact table

StoreID	...	ItemsSold	...
001		2000	

Store dimension table

StoreID	...	Size	...
001		250	



The size of a store expands

StoreID	...	ItemsSold	...
001		2000	

StoreID	...	Size	...
001		450	



A new fact arrives

StoreID	...	ItemsSold	...
001		2000	
001		3500	

StoreID	...	Size	...
001		450	

What's the problem here?

Solution 1-outline

- **Solution 1:** Overwrite the old values in the dimension tables
- Consequences
 - Old facts point to rows in the dimension tables with incorrect information!
 - New facts point to rows with correct information
- **Pros**
 - Easy to implement
 - Useful if the updated attribute is not significant, or the old value should be updated for error correction
- **Cons**
 - Old facts may point to “incorrect” rows in dimensions

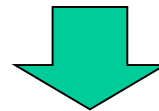
Solution 2

- **Solution 2: Versioning of rows with changing attributes**
 - The *key* that links dimension and fact table, identifies a *version* of a row, not just a “row”
 - **Surrogate keys make this easier to implement**
 - ◆ – what if we had used, e.g., the shop’s zip code as key?
 - ◆ Always use surrogate keys!!!
- **Consequences**
 - **Larger dimension tables**
- **Pros**
 - Correct information captured in DW
 - No problems when formulating queries
- **Cons**
 - Cannot capture the development over time of the subjects the dimensions describe
 - ◆ e.g., relationship between the old store and the new store not captured

Solution 2: Versioning of Rows

StoreID	...	ItemsSold	...
001		2000	

StoreID	...	Size	...
001		250	



different versions of a store

StoreID	...	ItemsSold	...
001		2000	

StoreID	...	Size	...
001		250	
002		450	



A new fact arrives

StoreID	...	ItemsSold	...
001		2000	
002		3500	

StoreID	...	Size	...
001		250	
002		450	

Which store has the old
fact (new fact) ?

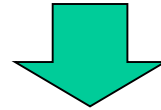
Solution 3

- **Solution 3: Create two versions of each changing attribute**
 - One attribute contains the current value
 - The other attribute contains the previous value
- Consequences
 - Two values are attached to each dimension row
- Pros
 - Possible to compare across the change in dimension value (which is a problem with Solution 2)
 - ◆ Such comparisons are interesting when we need to work simultaneously with two alternative values
 - ◆ Example: Categorization of stores and products
- Cons
 - Not possible to see when the old value changed to the new
 - Only possible to capture the two latest values

Solution 3: Two versions of Changing Attribute

StoreID	...	ItemsSold	...
001		2000	

StoreID	...	DistrictOld	DistrictNew	...
001		37	37	



versions of an attribute

StoreID	...	ItemsSold	...
001		2000	

StoreID	...	DistrictOld	DistrictNew	...
001		37	73	



StoreID	...	ItemsSold	...
001		2000	
001		3500	

StoreID	...	DistrictOld	DistrictNew	...
001		37	73	

We cannot find out **when**
the district changed.

Solution 2A

- **Solution 2A: Use special facts for capturing changes in dimensions via the Time dimension**
 - Assume that no simultaneous, new fact refers to the new dimension row
 - Insert a new special (null) fact that points to the new dimension row, and through its reference to the Time dimension, timestamps the row
- Pros
 - Possible to capture the development over time of the subjects that the dimensions describe
- Cons
 - Larger fact table

Solution 2A: Inserting Special Facts

StoreID	TimeID	...	ItemsSold	...
001	234		2000	

StoreID	...	Size	...
001		250	

special fact for capturing changes

StoreID	TimeID	...	ItemsSold	...
001	234		2000	
002	345		-	

StoreID	...	Size	...
001		250	
002		450	

StoreID	TimeID	...	ItemsSold	...
001	234		2000	
002	345		-	
002	456		3500	

StoreID	...	Size	...
001		250	
002		450	

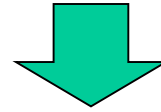
Solution 2B--Timestamping

- **Solution 2B:** Versioning of rows with changing attributes like in Solution 2 + timestamping of rows
- Pros
 - Correct information captured in DW
- Cons
 - Larger dimension tables

Solution 2B: Timestamping

StoreID	TimeID	...	ItemsSold	...
001	234		2000	

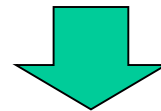
StoreID	Size	From	To
001	250	234	-



attributes: "From", "To"

StoreID	TimeID	...	ItemsSold	...
001	234		2000	

StoreID	Size	From	To
001	250	234	345
002	450	345	-



StoreID	TimeID	...	ItemsSold	...
001	234		2000	
002	456		3500	

StoreID	Size	From	To
001	250	234	345
002	450	345	-

Example of Using Solution 2B

- Product descriptions are versioned, when products are changed, e.g., new package sizes
 - Old versions are still in the stores, new facts can refer to both the newest and older versions of products
- However changes in Size for a store, where all facts from a certain point in time will refer to the newest Size value

Rapidly Changing Dimensions

- Difference between “slowly” and “rapidly” is subjective
 - Solution 2 is often still feasible
 - The problem is the size of the dimension
- Example
 - Assume an Employee dimension with 100,000 employees, each using 2K bytes and many changes every year
 - Solution 2B is recommended
- Examples of (large) dimensions with many changes: Product and Customer
- The more attributes in a dimension table, the more changes per row are expected
- Example
 - A Customer dimension with 100M customers and many attributes
 - Solution 2 yields a dimension that is too large

Solution 4: Dimension Splitting

Customer dimension (original)

CustID
Name
PostalAddress
Gender
DateofBirth
Customerside
...
NoKids
MaritalStatus
CreditScore
BuyingStatus
Income
Education
...



CustID
Name
PostalAddress
Gender
DateofBirth
Customerside
...

Customer dimension (new):

relatively static attributes



DemographyID
NoKids
MaritalStatus
CreditScoreGroup
BuyingStatusGroup
IncomeGroup
EducationGroup
...

Demographics dimension:

often-changing attributes

Solution 4--Outline

- Solution 4
 - Make a “minidimension” with the often-changing (demographic) attributes
 - Convert (numeric) attributes with many possible values into attributes with few discrete or banded values
 - ◆ E.g., Income group: [0,10K), [0,20K), [0,30K), [0,40K)
 - ◆ **Why? Any Information Loss?**
 - Insert rows for all combinations of values from these new domains
 - ◆ With 6 attributes with 10 possible values each, the dimension gets $10^6=1,000,000$ rows
 - If the minidimension is too large, it can be further split into more minidimensions
 - ◆ Here, synchronous/correlated attributes must be considered (and placed in the same minidimension)
 - ◆ The same attribute can be repeated in another minidimensions

Solution 4 (Changing Dimensions)

- **Pros**

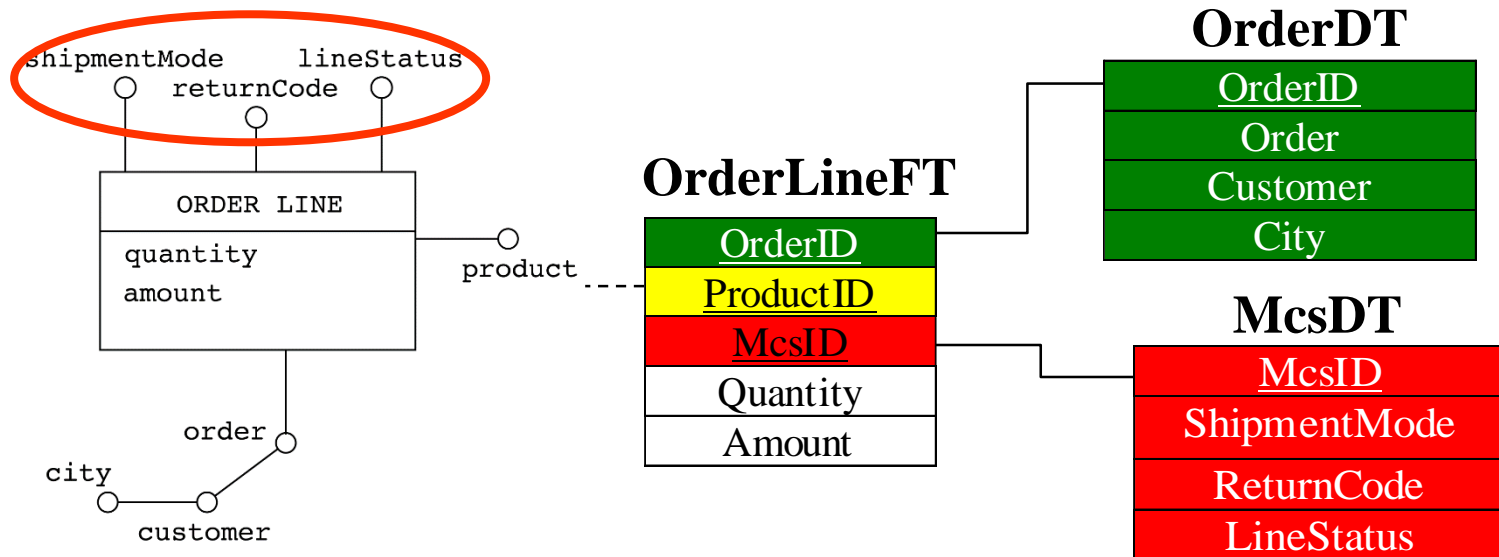
- DW size (dimension tables) is kept down
- Changes in a customer's demographic values do not result in changes in dimensions

- **Cons**

- More dimensions and more keys in the star schema
- Navigation of customer attributes is more cumbersome as these are in more than one dimension
- Using value groups gives less detail
- The construction of groups is irreversible

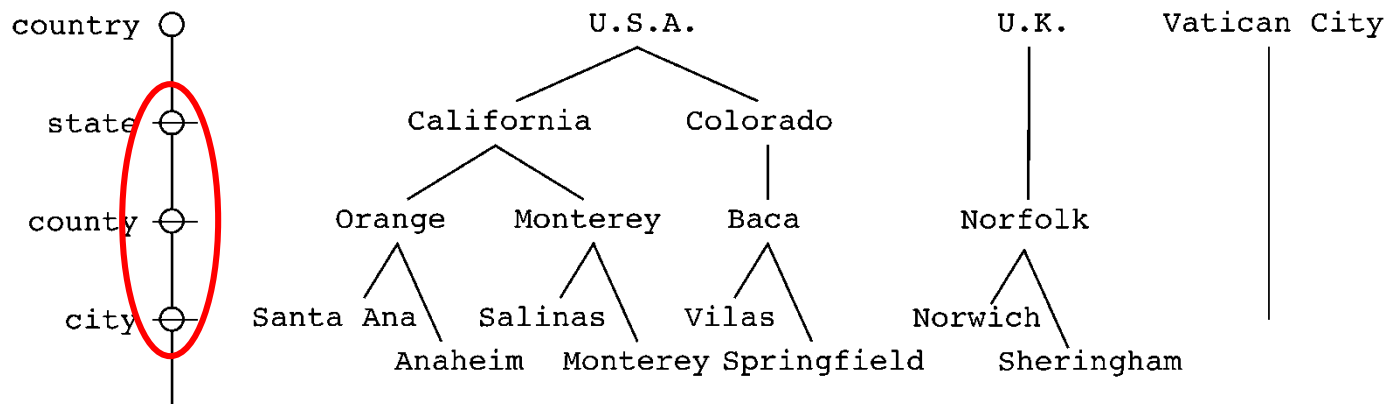
Degenerate dimensions

- This term refers to a hierarchy including only one attribute
 - If the attribute is not too long, its values can be directly included in the FT
 - Alternatively, we may use one DT to model several degenerate dimensions (*junk dimension*)
 - ◆ Within a junk dimension there is no functional dependency between attributes, so all combination of values are valid
 - ◆ This solution is feasible only if the number of distinct values for the attributes involved is small



Incomplete hierarchies

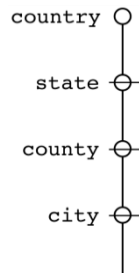
- One or more aggregation levels are not available for some instances
- You can model incomplete hierarchies if you enter fake values into dimension tables
- The lack of a value at a certain aggregation level does not imply that no values at higher aggregation levels exist, so keeping roll-up consistent may become a problem



- Three solutions are possible: they differ in the values entered as a signpost for missing attribute values

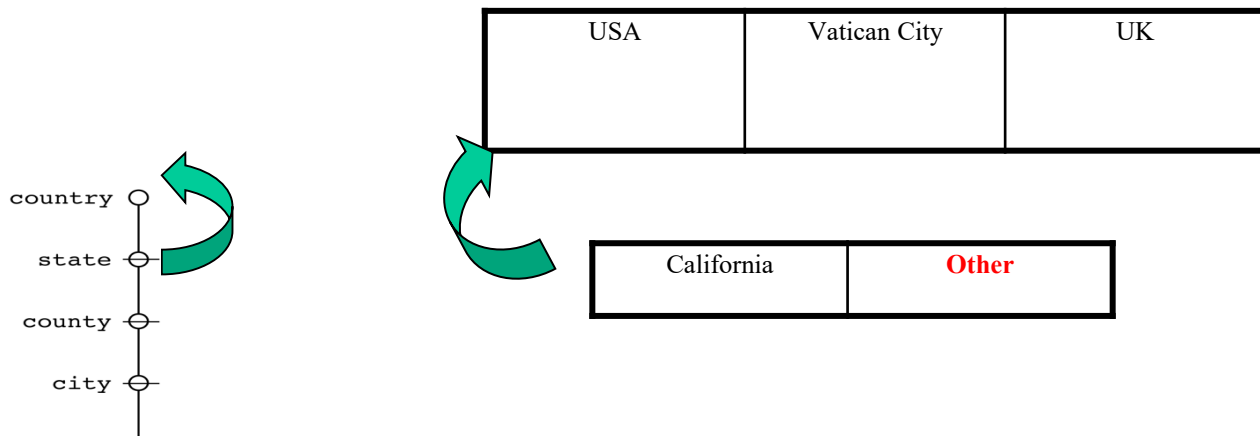
Incomplete hierarchies

- **Balancing by exclusion:** Missing values are replaced by a single generic signpost, such as 'other'



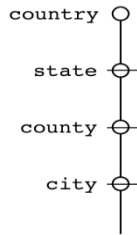
USA	Vatican City	UK	UK
California	Other	Other	Other
Orange	Other	Norfolk	Essex
Santa Ana	Other	Norwich	Epping

- Preferable if many instances have missing values
- **The regular roll-up semantics is broken:** if you further aggregate data, a number of groups collapse into one single group detail



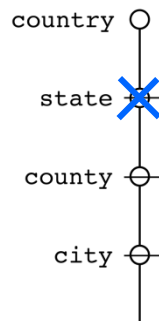
Incomplete hierarchies

- **Downward balancing:** Missing values are replaced by the value of the attribute that precedes it in its hierarchy .



USA	Vatican City	UK	UK
California	Vatican City	Norfolk	Essex
Orange	Vatican City	Norfolk	Essex
Santa Ana	Vatican City	Norwich	Epping

- Preferable when there is only a very limited amount of missing data
- **The interpretation of the final results is complicated because some attribute values will also appear at the wrong aggregation level**



California	Vatican City	Norfolk	Essex
------------	--------------	---------	-------