COMP810 Data Warehousing and Big Data

Semester 2 2024

Dr Victor Miranda

COMP810

Week 3 Data Warehousing

- Logical Model – II (Snowflake, Constellation schemas)
- SQL operations

COMP810

Heads up: Week 4 Data Warehousing

- Database Diagram Design Tools
- SQL operations

Week 2 (Week 8) Summary

- OLTP vs OLAP
- Logical Model - Star Schema
- SQL operations

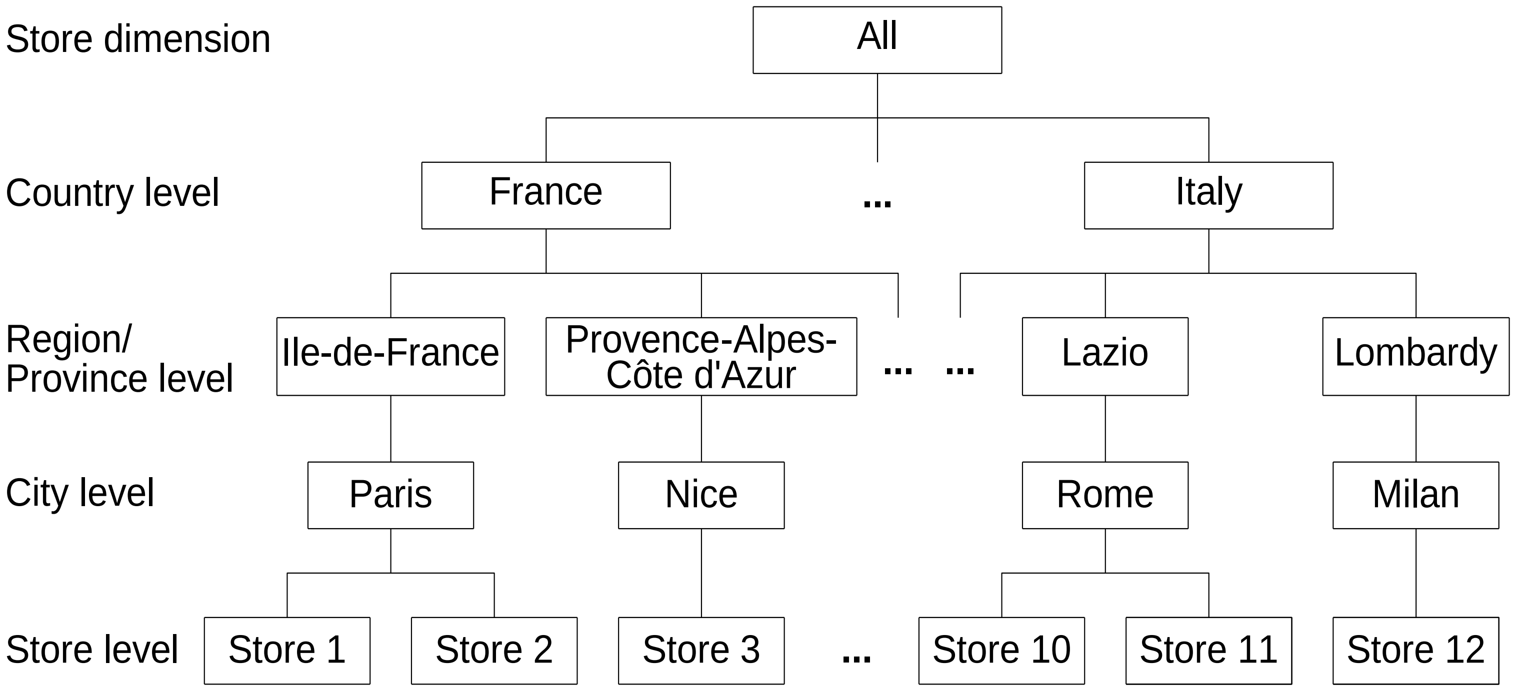
Re .DW Project

- Section A – After today's lecture
- Section B – Now..gradually

Dimensions are hierarchically organized

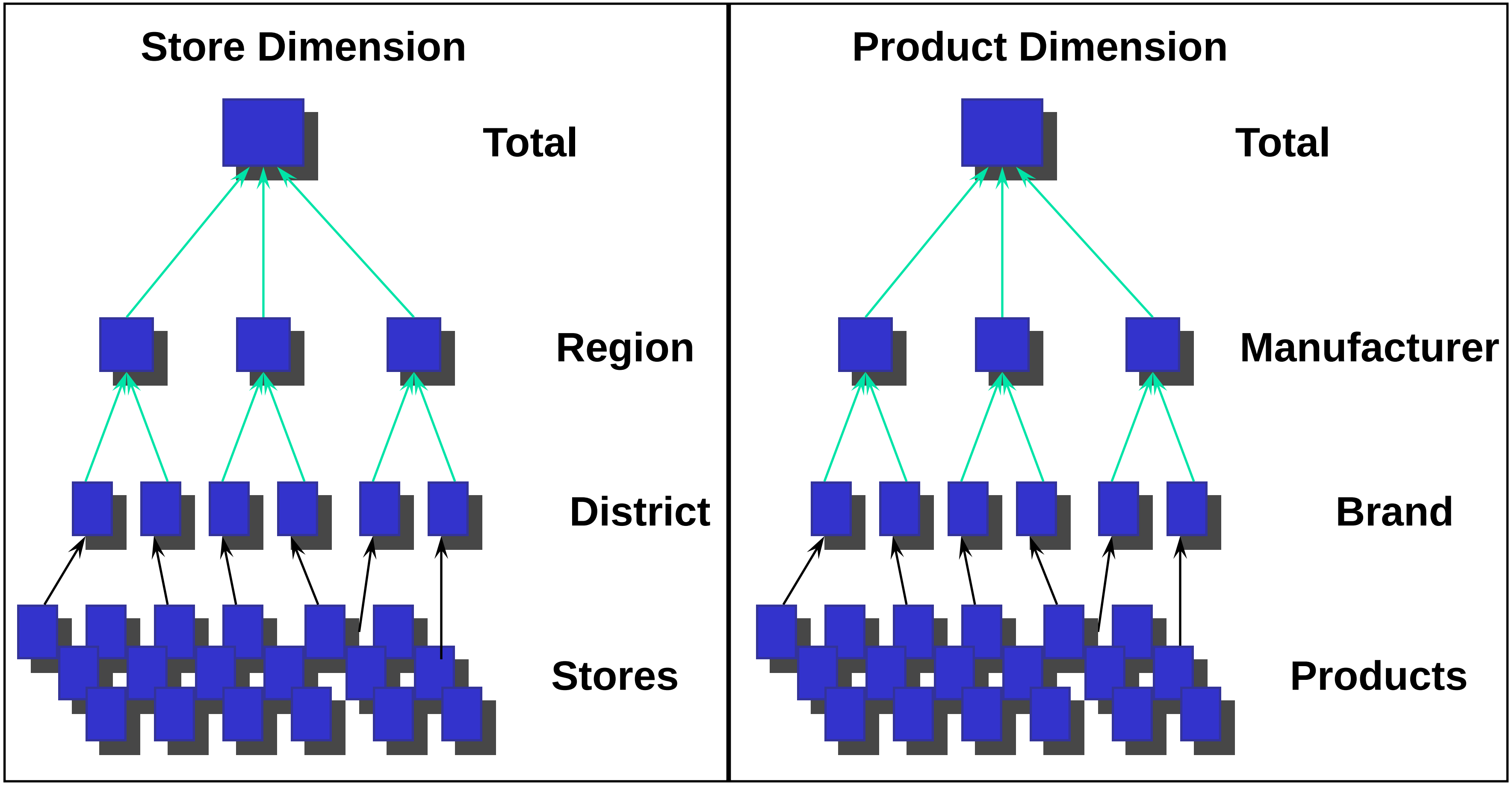
- Data granularity: Level of detail of measures
- Data analyzed at different granularities (abstraction levels)
- Hierarchies relate low-level (detailed) concepts to higher-level (general concepts)
  - Example: Store – City – Region/Province – Country
- Given two related levels in a hierarchy, lower level is called child, higher level is called parent
- Instances of these levels are called members

Kind of Hierarchies



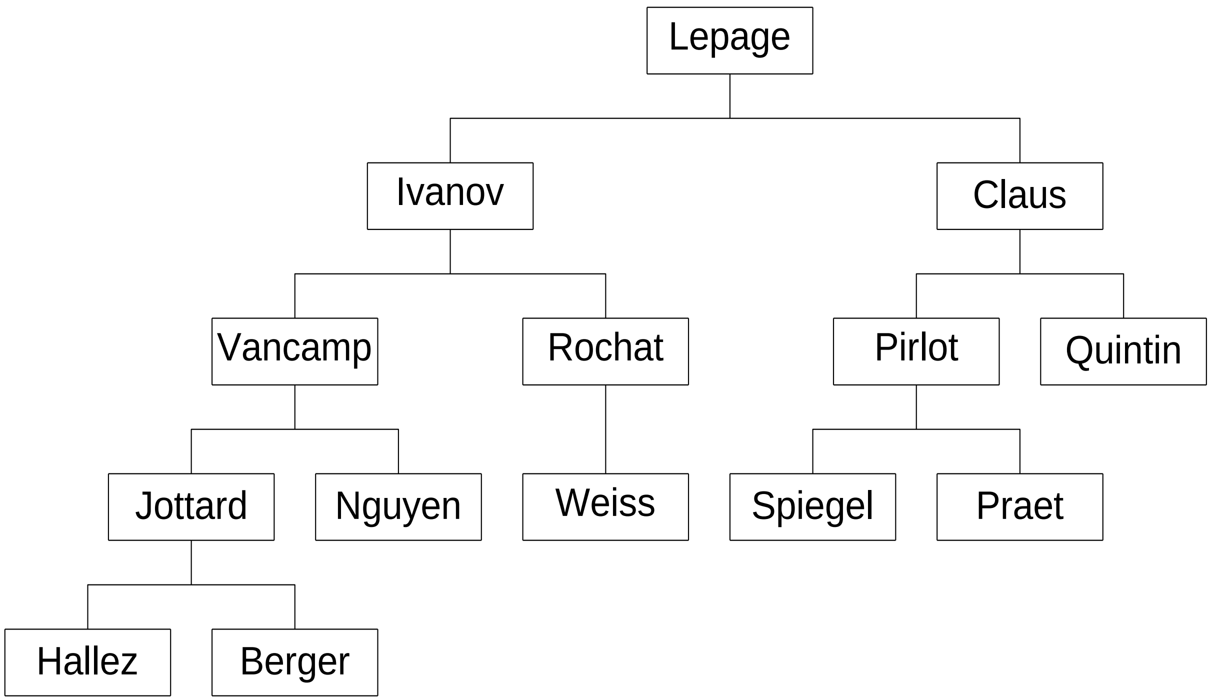
- Balanced: Same number of levels from leaf members to root
- Noncovering: Parent of some members does not belong to the level immediately above
  - E.g., small countries may not have Region/Province level

Kinds of Hierarchies (Example 2 – Balancer)



Kinds of Hierarchies

- Unbalanced: May not have instances at the lower levels
  - E.g., some cities do not have stores, sales are through wholesalers
- Parent-child (recursive): Involve twice the same level



Conceptual Modelling of Data Warehouses

- Modelling data Warehouses –

- Star schema: One fact table and a set of dimension tables
  - Referential integrity constraints between fact table and dimension tables
  - Dimension tables may contain redundancy in the presence of hierarchies

- Snowflake schema: Avoids redundancy of star schemas by normalizing dimension tables
  - Normalized tables optimize storage space, but decrease performance

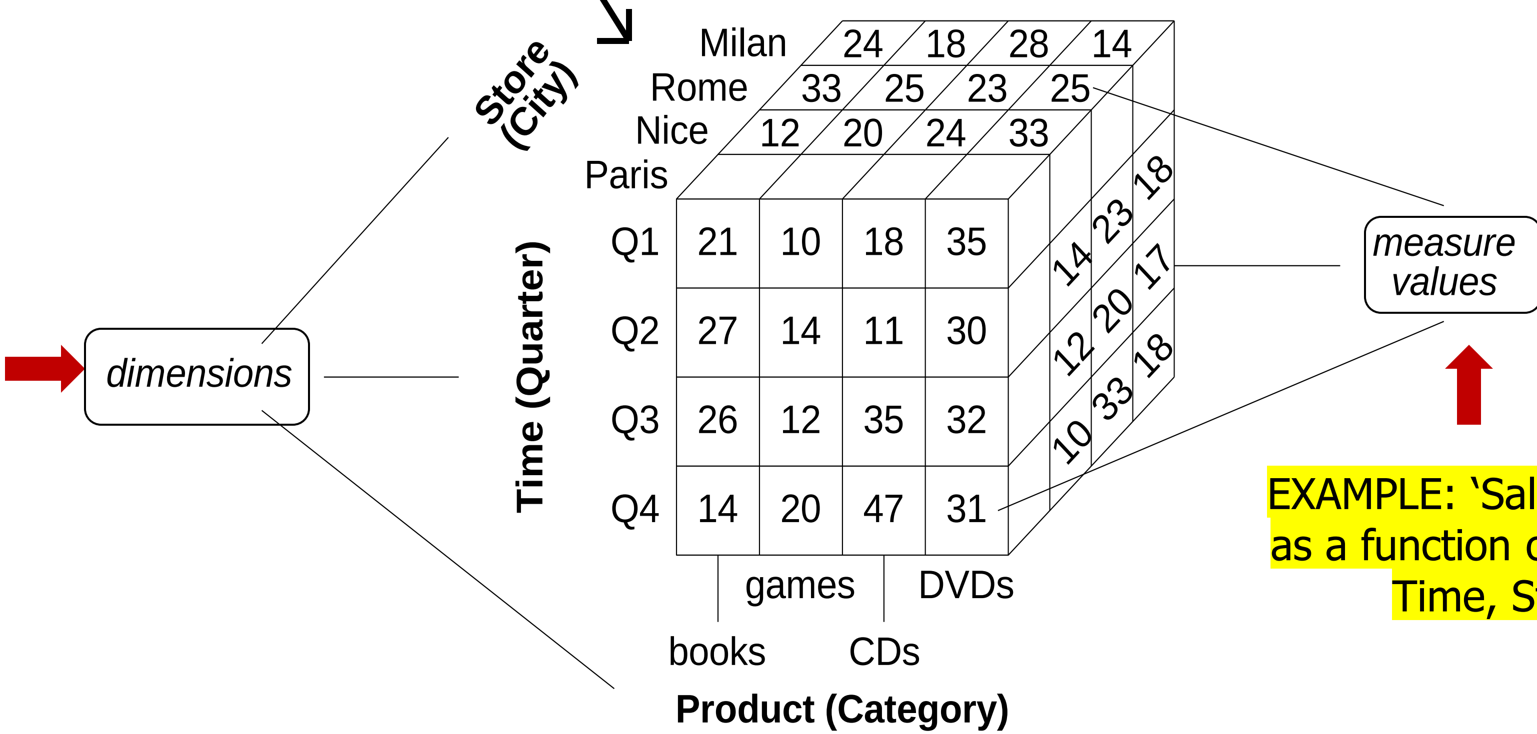
- Constellation schema: Multiple fact tables that share dimension tables

Conceptual Modelling of Data Warehouses

Dimensional model vs Multidimensional view of data

Every data warehouse can be seen as a multidimensional data model represented as a **data cube** or a **hypercube**

- Dimensions: Perspectives for analyzing data
- Cells (facts): Contain measures, values that are to be analyzed



EXAMPLE: 'Sales volume' as a function of product, Time, Store

Snowflake schema

Dimensional models are more denormalized and optimized for data querying, while normalized models *seek to eliminate data redundancies*

Snowflake schema

Dimensional models are more denormalized and optimized for data querying, while normalized models *seek to eliminate data redundancies*

Denormalization and normalization are two opposite ways of organizing data in a relational database.

- Denormalization means combining data from multiple tables into one, reducing the number of joins and improving query performance.

- Normalization means splitting data into multiple tables, eliminating redundancy and ensuring data integrity.

Snowflake schema

Dimensional models are more denormalized and optimized for data querying, while normalized models *seek to eliminate data redundancies*

- Snowflake schema: Avoids redundancy of star schemas by normalizing dimension tables
  - Normalized tables optimize storage space, but decrease performance

Logical structure of the model - Star Schema

E1

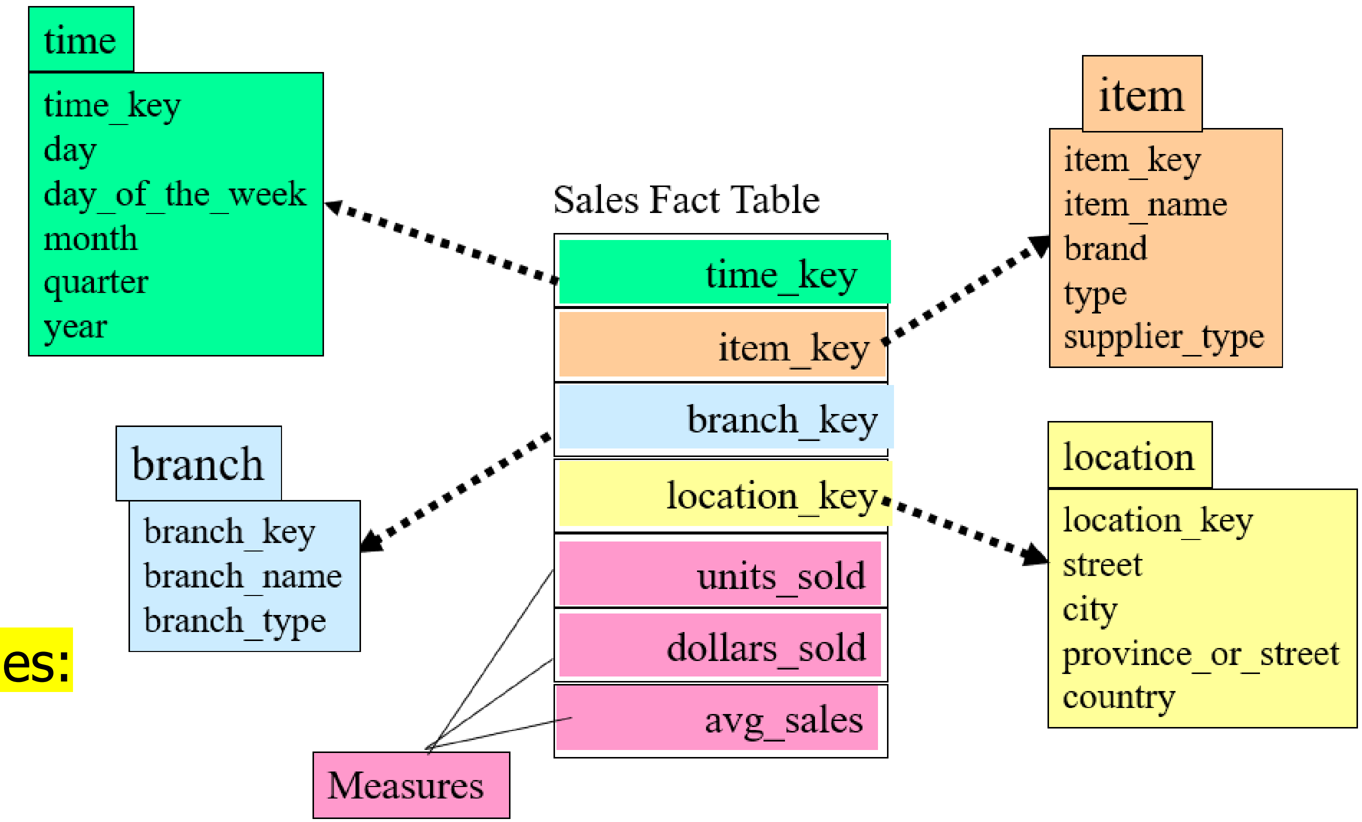
Fact: Sales

Measures:

- Units_sold
- dollars_sold
- avg_sales

Dimension tables:

Time,
Branch,
item,
Location



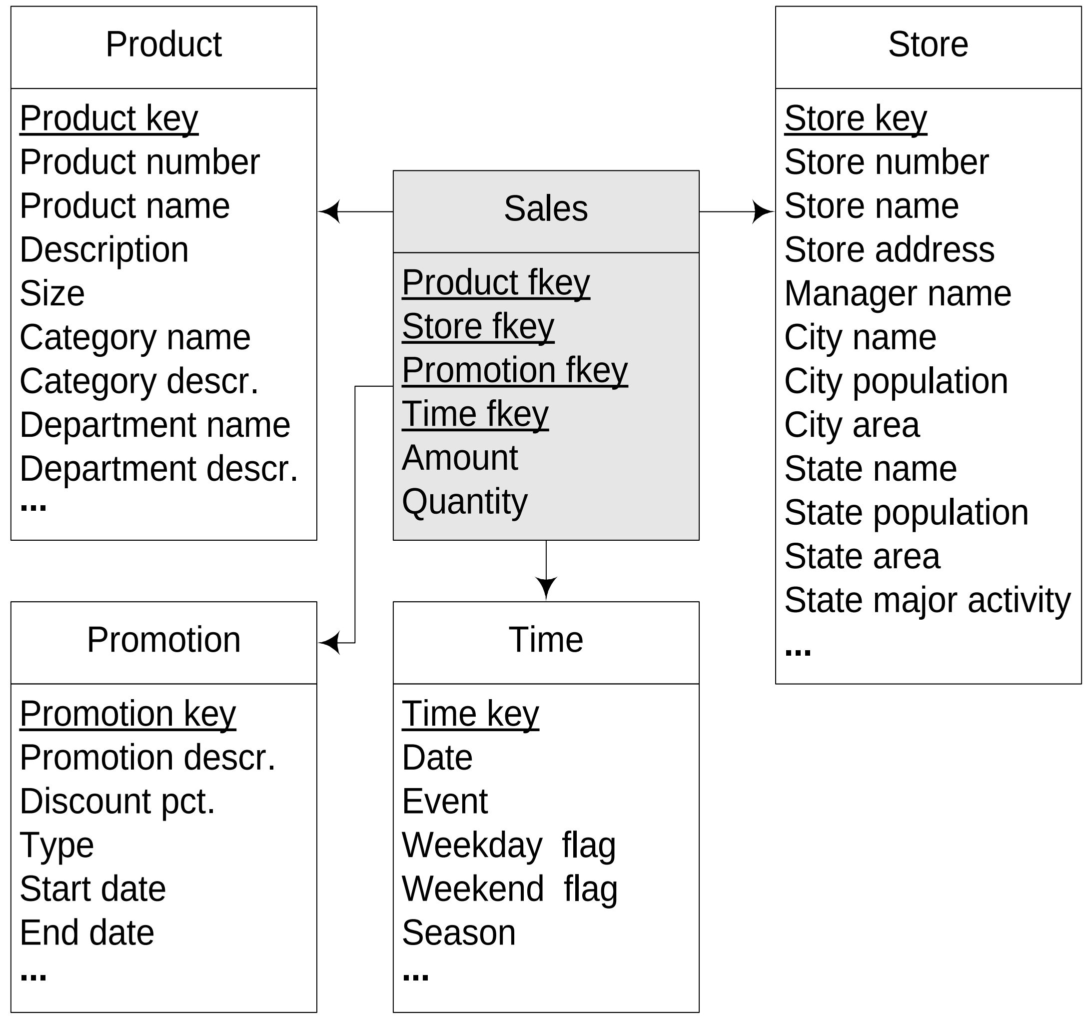
Logical structure of the model - Star Schema

E2

Fact: Sales

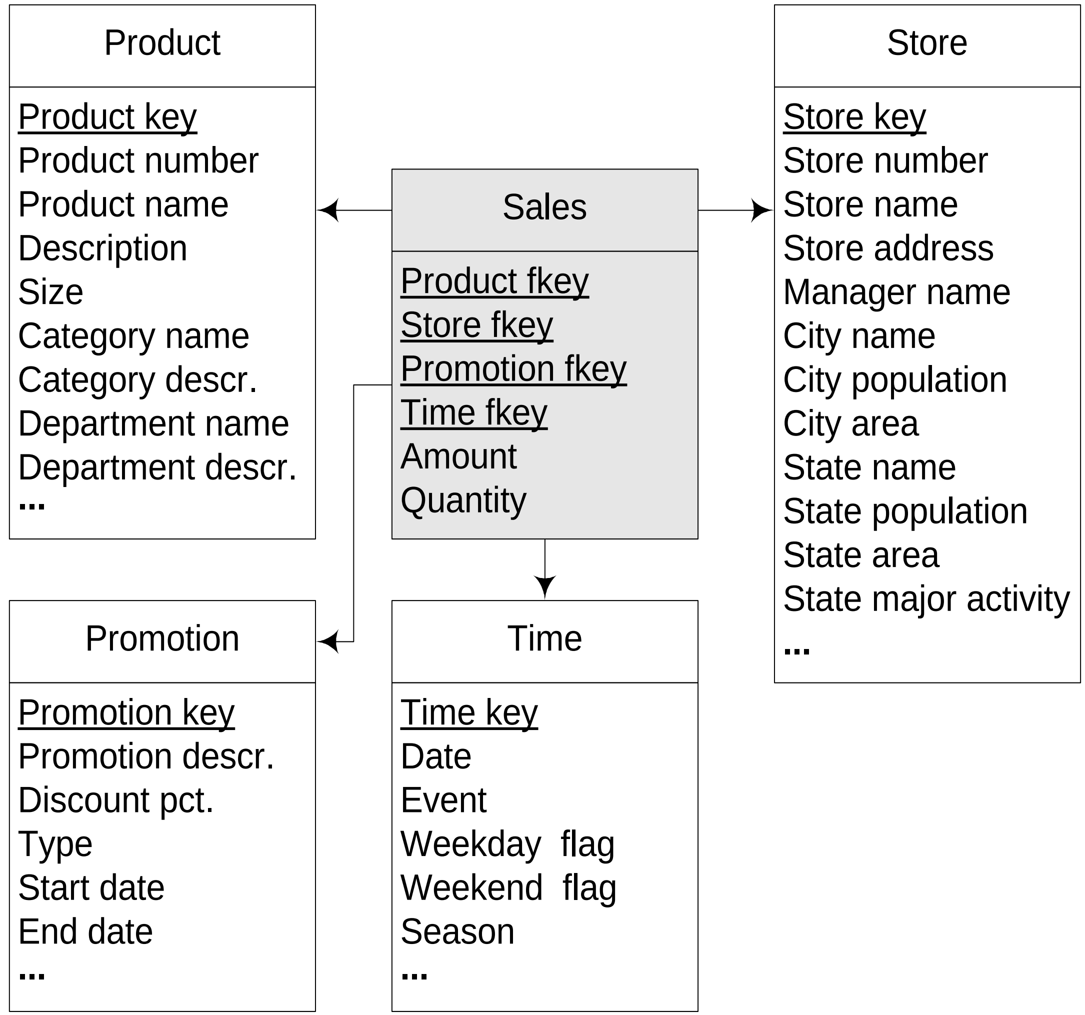
Measures:
Amount,
Quantity

Dimension tables:
Product,
Promotion,
Time,
Location

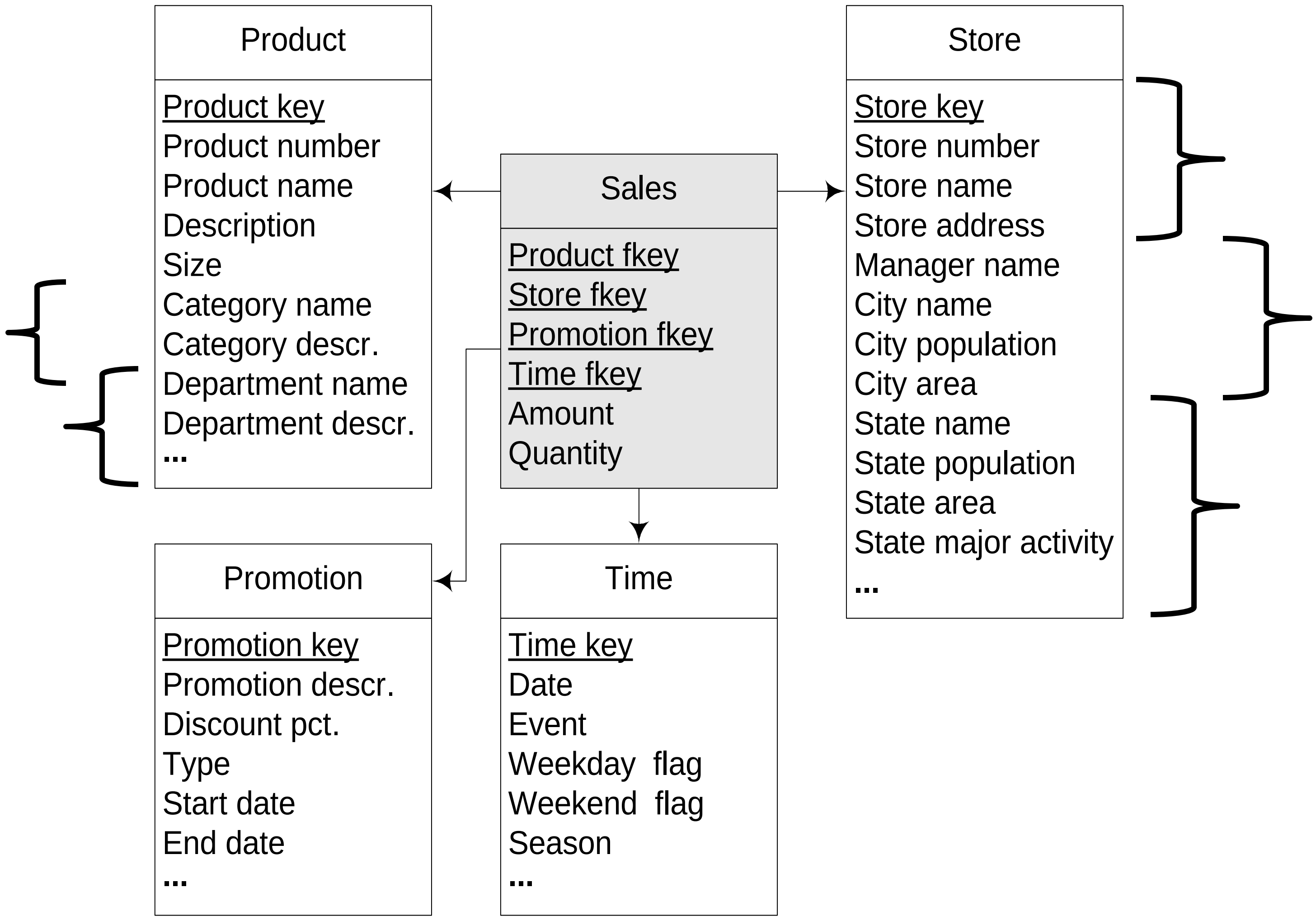


Normalize E1 - keeping the only Fact table

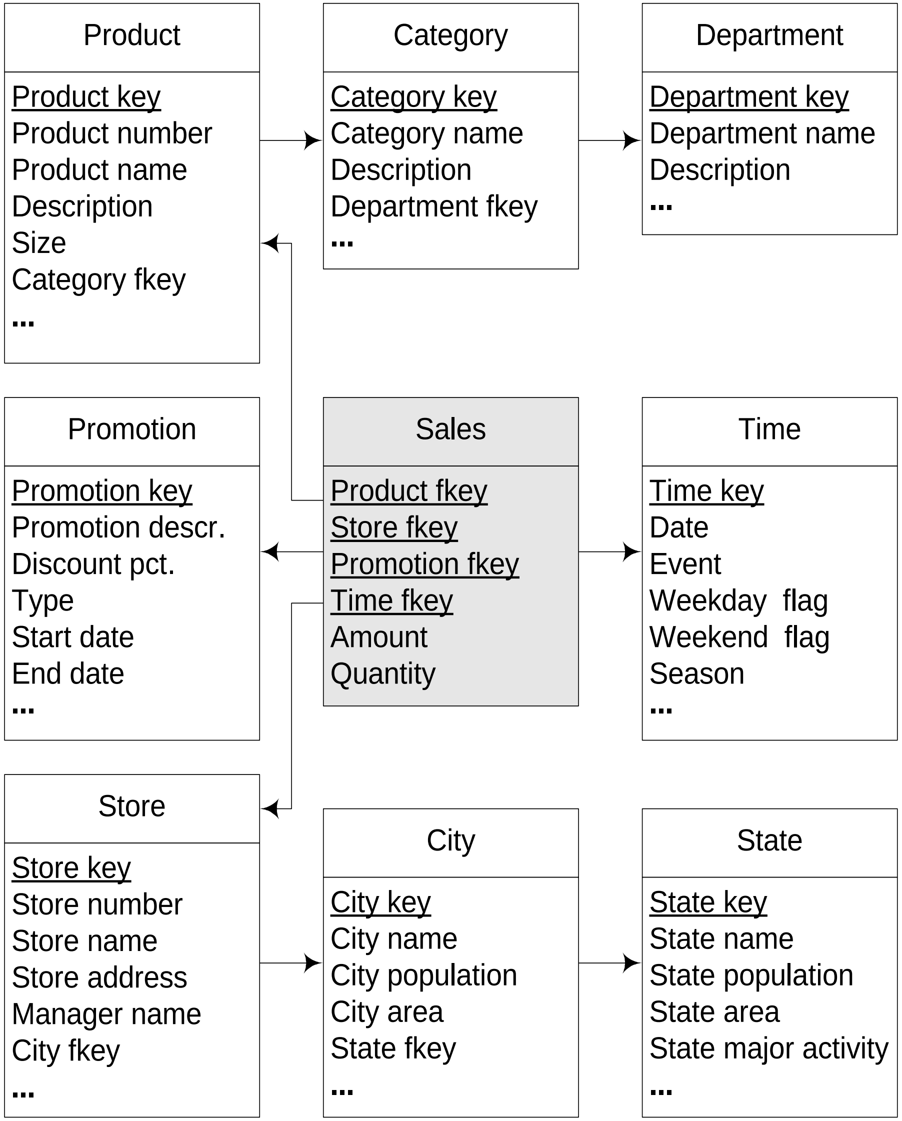
There is NO definitive answer



Normalize E1 …



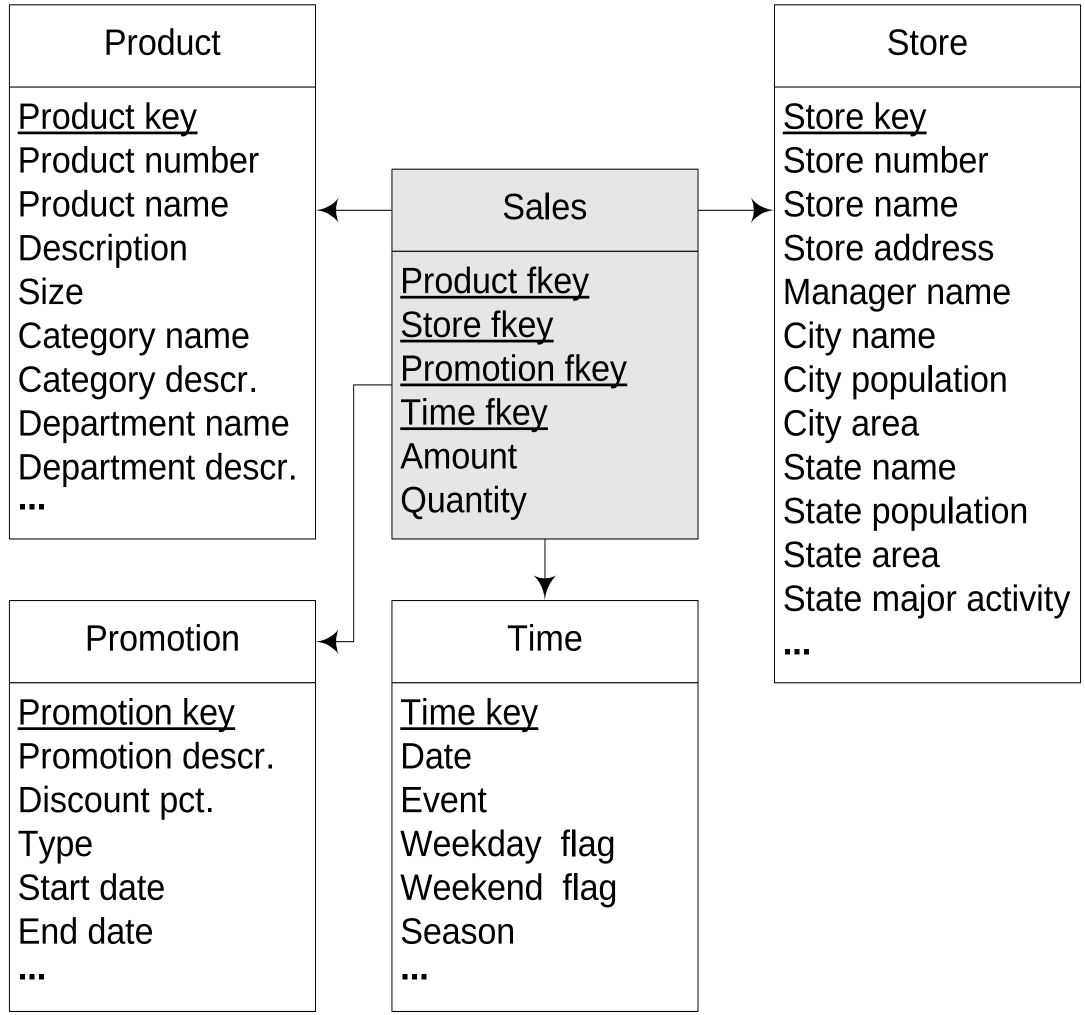
Logical DW Design: Snowflake Schema (answer)



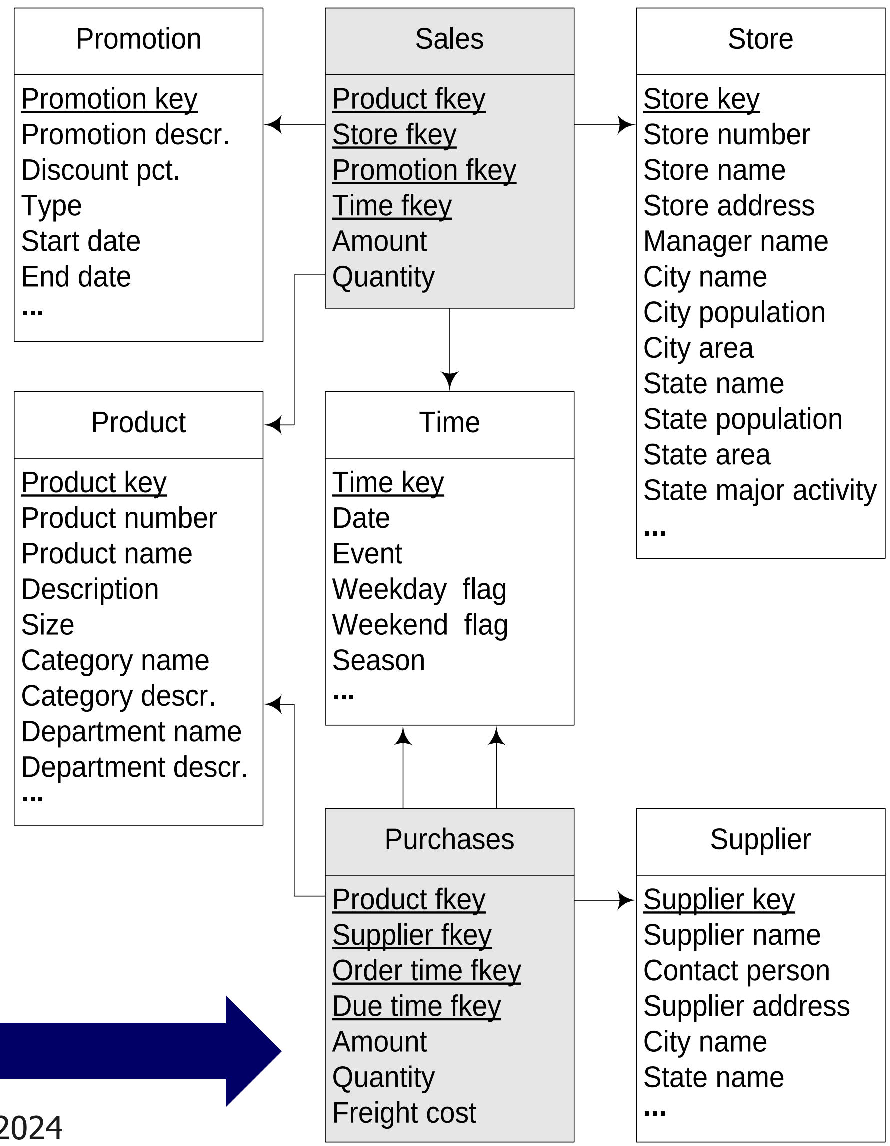
Now, we need to aggregate the freight cost.

There is NO definitive answer

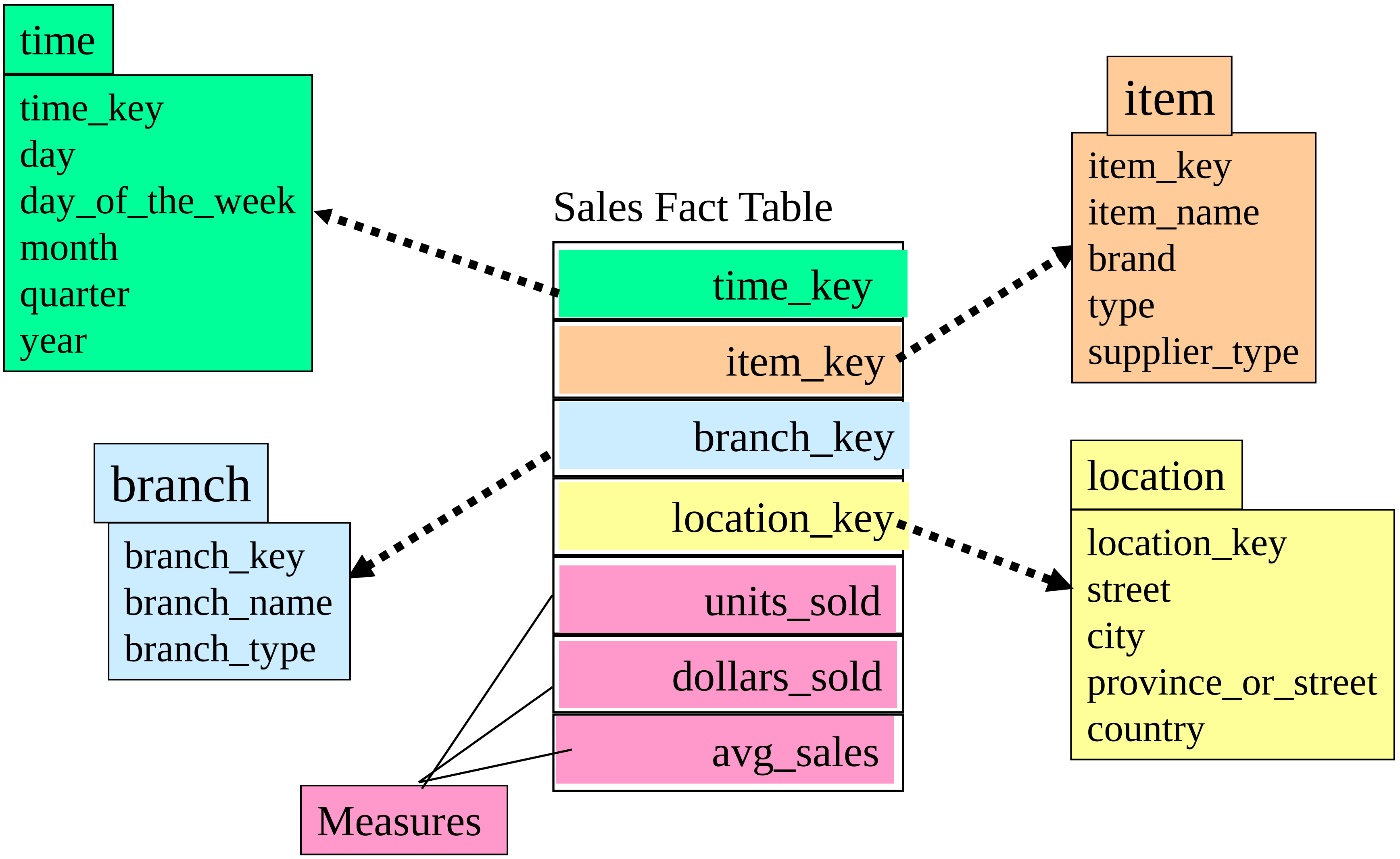
Hint: freight cost also depends on ‘amount’ and ‘quantity’



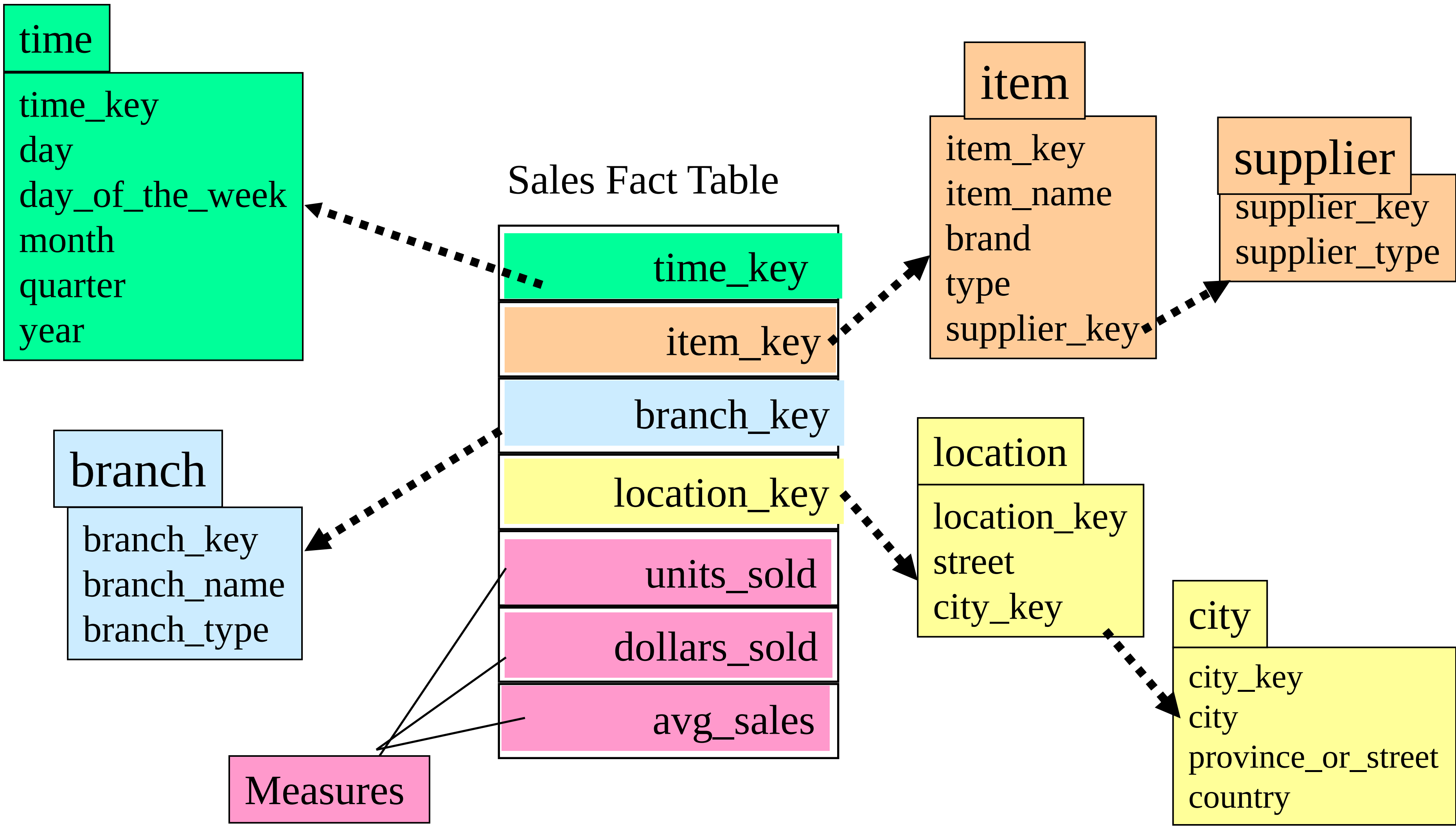
Logical DW Design: Fact Constellation Schema !



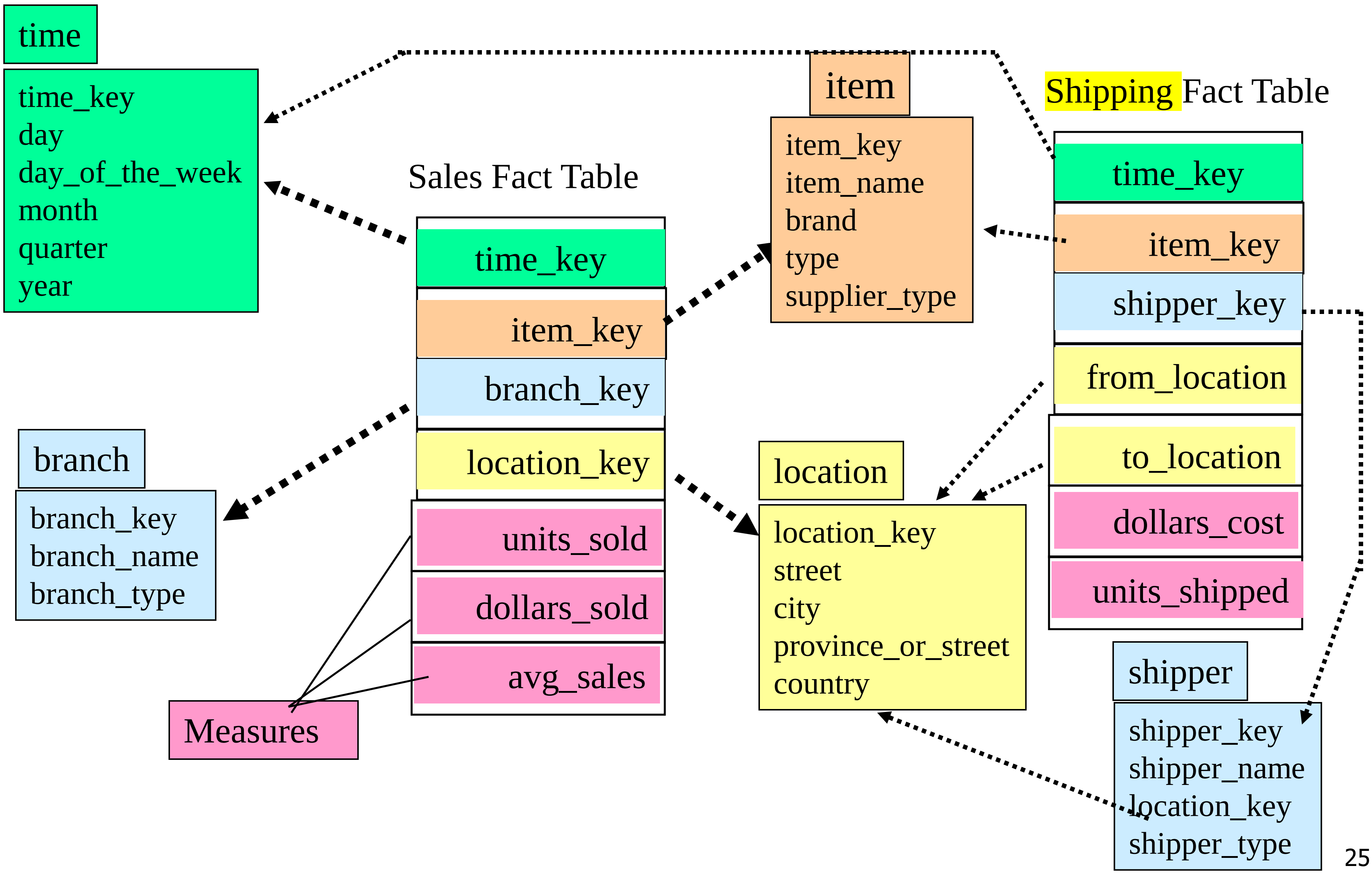
Another Example of Star Schema



Example of Snowflake Schema



Example of Fact Constellation



How to balance denormalization and normalization?

A trade-off: There is NO definitive answer, as it depends on factors such as data volume, business requirements, query patterns, and performance expectations.

Hint 1: When denormalizing, avoid repeating large or complex attributes that may change frequently or cause data inconsistency

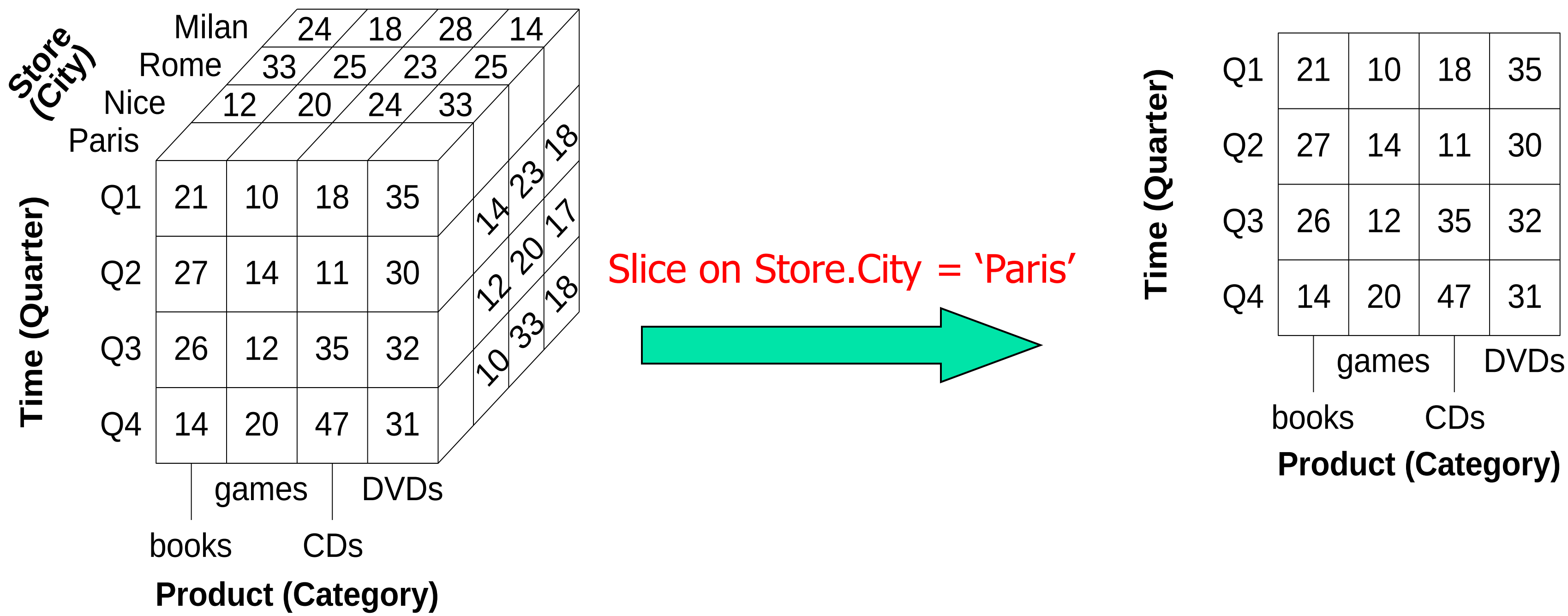
Hint 2: When normalizing, avoid creating too many or too narrow tables that increase complexity and the number of joins

Operations in Multidimensional Data Model

- Selection (Slice)
- Projection
- Aggregation (roll- up)
- Navigation (drill down)

Selection (slices)

- Performs a selection on one dimension of a cube, resulting in a subcube (in practice, select tuples or rows)



In practice, select tuples

Select students with gpa higher than 3.3 from S1*:*

$$\sigma_{gpa>3.3}(S1)$$

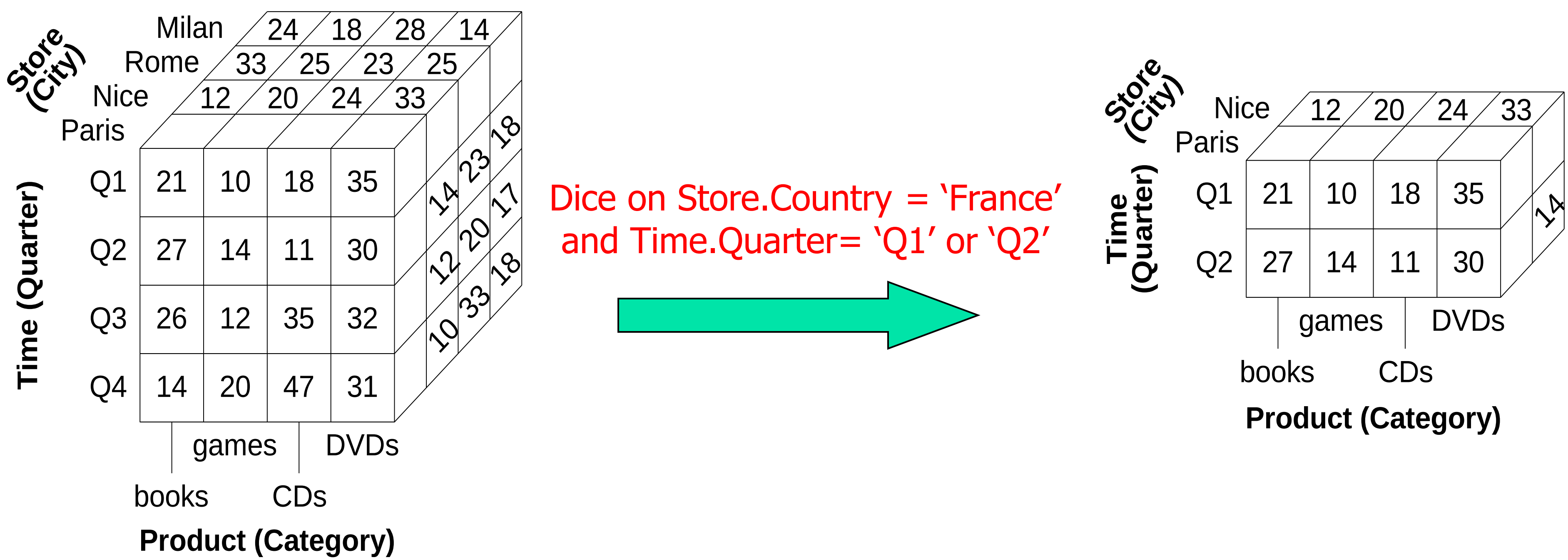
S1

sid	name	gpa
50000	Dave	3.3
53666	Jones	3.4
53688	Smith	3.2
53650	Smith	3.8
53831	Madayan	1.8
53832	Guldu	2.0

sid	name	gpa
53666	Jones	3.4
53650	Smith	3.8

Projection

- Defines a selection on two or more dimensions, thus again defining a subcube (in practice, select columns)



In practice, select columns

Project name and gpa of all students in S1:

$\Pi_{\text{name, }gpa}(S1)$

S1

Sid	name	gpa
50000	Dave	3.3
53666	Jones	3.4
53688	Smith	3.2
53650	Smith	3.8
53831	Madayan	1.8
53832	Guldu	2.0

name	gpa
Dave	3.3
Jones	3.4
Smith	3.2
Smith	3.8
Madayan	1.8
Guldu	2.0

Combine Selection and Projection

- Project name and gpa of students in S1 with gpa higher than 3.3:

$$\Pi_{\text{name},gpa}(\sigma_{gpa>3.3}(\text{S1}))$$

Sid	name	gpa
50000	Dave	3.3
53666	Jones	3.4
53688	Smith	3.2
53650	Smith	3.8
53831	Madayan	1.8
53832	Guldu	2.0

name	gpa
Jones	3.4
Smith	3.8

SQL Query

Basic form: (plus many many more bells and whistles)

```
SELECT  <attributes>
FROM    <one or more relations>
WHERE   <conditions>
```

Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   *
FROM     Product
WHERE    category='Gadgets'
```

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks

"selection"

Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   PName, Price, Manufacturer
FROM     Product
WHERE    Price > 100
```

PName	Price	Manufacturer
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

"selection" and "projection"

Aggregation

- Many OLAP operations are **based** on a fact table AND involve ***aggregation*** of the data in the fact table

SQL is excellent at aggregating data

> 'aggregate' functions we'll cover today:

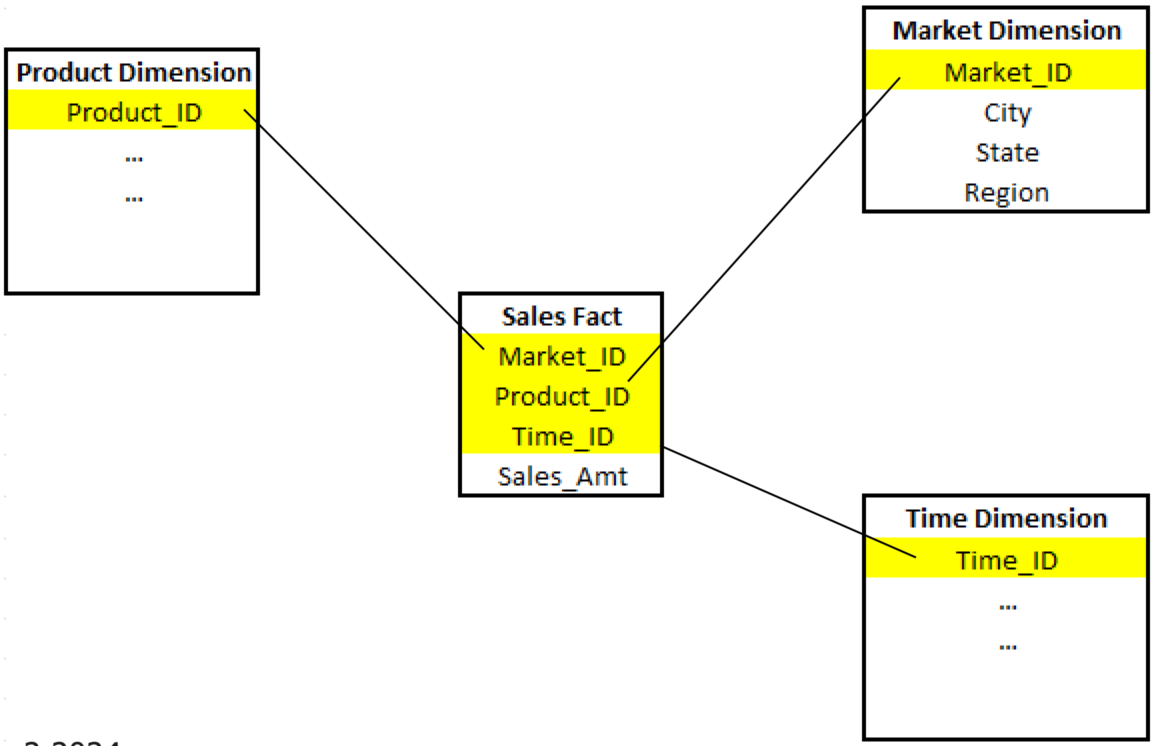
- `COUNT` counts the # of rows in a particular column.
- `SUM` adds together all the values in a particular column.
- `MIN` and `MAX` return the lowest and highest values in a particular column, respectively.
- `AVG` calculates the average of a group of selected values.

Except count, all aggregations apply to a single attribute

Consider:

Sales (*Market_Id, Product_Id, Time_Id, Sales_Amt*)

Market (*Market_Id, City, State, Region*)



Aggregation (count)

`COUNT` is an SQL aggregate function for counting the number of rows in a particular column

Example 1:

```
SELECT COUNT(*)
FROM Sales ;
```

Gives the number of rows of the 'Sales' table

Aggregation (count)

`COUNT` is an SQL aggregate function for counting the number of rows in a particular column

Example 1:

SELECT COUNT(*)
FROM *Sales* ;

Same as:

SELECT *
FROM *Sales ;* ??

Aggregation (count)

`COUNT` is an SQL aggregate function for counting the number of rows in a particular column

Example 2:

SELECT COUNT(*Sales_Amt*)
FROM *Sales*;

Gives the count of all rows where 'Sales_Amt' is different from NULL.

Is this number < = > than Example 1?

SELECT COUNT(*)
FROM *Sales* ;

Aggregation (count)

`COUNT` is an SQL aggregate function for counting the number of rows in a particular column

Example 2:

SELECT COUNT(*Sales_Amt*)
FROM *Sales*;

Gives the count of all rows where 'Sales_Amt' is different from NULL.
This number IS NOT greater than that from Example 1 as *Sales_Amt* may have some NULLs.

Aggregation (count)

`COUNT` on counting non-numerical columns.

Example 3:

SELECT COUNT(*Market_Id*)
FROM *Sales* ;

Aggregation (count)

`COUNT` on counting non-numerical columns.

Example 3:

SELECT COUNT(*Market_Id*)
FROM *Sales* ;

Same (<) result as Example 1.
`COUNT` simply counts the total number of non—null rows, not the distinct values.

Aggregation (count) - Housekeeping

The column header in the results just reads "count."

Consider naming your columns so that they make a little more sense to anyone else who views your work.

Example:

SELECT COUNT(*Sales_Amt*) as “count of Sales”

FROM *Sales* ;

or

SELECT COUNT(*Sales_Amt*) as count_of_sales

FROM *Sales* ;

Aggregation (sum)

Aggregation (sum)

The SQL `sum` function gives the 'totals' from a given column.

Example:

```
SELECT SUM(Sales_Amt)
FROM Sales ;
```

Aggregation (sum)

The SQL `sum` function gives the 'totals' from a given column.

Example:

```
SELECT SUM(Sales_Amt)
FROM Sales ;
```

The query above returns the total of the *Sales_Amt* column.

As opposed to `count`, with `sum` NULLs are treated as zeros.

The SQL MIN and MAX functions

Aggregation (min and max)

The SQL `min/max` functions return the lowest and highest values in a particular column.

- Similar to COUNT - they can be used on non-numerical columns.
  - Depending on the column type, MIN will return the lowest number, earliest date, or non-numerical value as close alphabetically to "A" as possible.

Aggregation (min and max)

The SQL `min/max` functions return the lowest and highest values in a particular column.

- Similar to COUNT - they can be used on non-numerical columns.
  - Depending on the column type, MIN will return the lowest number, earliest date, or non-numerical value as close alphabetically to "A" as possible.
- **MAX does the opposite—it returns the highest number, the latest date, or the non-numerical value closest alphabetically to "Z."**

Aggregation (min and max)

The following query returns the MIN and the MAX from the numerical column *Sales_Amt*

Example:

SELECT MIN(*Sales_Amt*) as “Sales Minimum” ,
MAX(*Sales_Amt*) as “Sales Maximum”
FROM *Sales* ;

As opposed to `count`, with `sum` NULLs are treated as zeros.

The SQL AVG function

Aggregation (avg)

The SQL `avg` function gives the average of a selected group of values. Can be used only in numerical columns

Example:

SELECT AVG(*Sales_Amt*)
FROM *Sales* ;

Aggregation (avg)

The SQL `avg` function gives the average of a selected group of values. IGNORES NULLs

Both queries below give the same output:

```
SELECT AVG(Sales_Amt)
FROM Sales ;
```

```
SELECT AVG(Sales_Amt)
FROM Sales
WHERE Sales_Amt IS NOT NULL;
```

A few more examples…

Table name

Attribute names

Tables in SQL

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Tuples or rows

Aggregation (count)

COUNT applies to duplicates, unless otherwise stated:

```
SELECT  Count(category)
FROM    Product
WHERE   year > 1995
```

same as Count(*)

We probably want:

```
SELECT  Count(DISTINCT category)
FROM    Product
WHERE   year > 1995
```

More examples…

Purchase(product, date, price, quantity)

```
SELECT  Sum(price * quantity)
FROM    Purchase
```

```
SELECT  Sum(price * quantity)
FROM    Purchase
WHERE   product = 'bagel'
```

What do they mean ?

Simple Aggregations

Purchase

Product	Date	Price	Quantity
Bagel	10/21	1	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Bagel	10/25	1.50	20

```
SELECT  Sum(price * quantity)
FROM    Purchase
WHERE   product = ‘bagel’
```

50 (= 20+30)

Q: Revenue from bagel sales?

Purchase

Product	Date	Price	Quantity
Bagel	10/21	1	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Bagel	10/25	1.50	20

```
SELECT  Sum(price * quantity)
FROM    Purchase
WHERE   product = 'bagel'
```

→ 50 (= 20+30)

Grouping and Aggregation

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```
SELECT      product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE       date > '10/1/2005'
GROUP BY    product
```

Let's see what this means…

COUNT, AVG, SUM in SQL

SQL aggregate function like COUNT, AVG, and SUM aggregate across the entire table.

> What if you want to aggregate only part of a table?

For example, you might want to count the number of entries for each year.

Store_ID	Item_ID	Day_ID	Revenue
1	10	3/9/99	16,152
1	10	3/10/99	28,541
1	10	3/11/99	80,892
1	10	3/12/99	51,200
...	...	...	...

Grouping and Aggregation

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUPBY

3. Compute the SELECT clause: grouped attributes and aggregates.

>> GROUP BY groups rows that have the same values into summary rows

Many OLAP queries involve ***aggregation*** of the data in the fact table across different dimensions

1&2. FROM-WHERE-GROUPBY

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

3. SELECT

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

Product	TotalSales
Bagel	50
Banana	15

```
SELECT      product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE       date > ‘10/1/2005’
GROUP BY    product
```

A few more examples…

Consider the table Sales:

Product ID	Market_ID	Sales_Amt
P1	M1	4567
P1	M1	3455
P1	M2	...
P1	M2	...
P1	M2	...
P2	M1	...
P2	M1	...
P2	M1	...
P3	M1	...
P3	M1	
P3	M2	
P3	M2	
P3	M3	
P3	M3	
P4	....	

Aggregation

Many OLAP queries involve ***aggregation*** of the data in the fact table across different dimensions

- For example, to find the total sales (over time) of each product, we might use

SELECT S.*Product_Id*, SUM (S.*Sales_Amt*)
FROM Sales S
GROUP BY S.*Product_Id ;*

Identifier

Aggregation (multidimensional model)

- The output of the previous query

Product_Id	SUM(*Sales_Amt*)
P1	4005
P2	6003
P3	4503
P4	7503
P5	…

Aggregation

Many OLAP queries involve ***aggregation*** of the data in the fact table

- However, to find the total sales (over time) of each product in each market, we use:

```
SELECT      S.Market_Id, S.Product_Id, SUM (S.Sales_Amt)
FROM        Sales  S
GROUP BY  S.Market_Id, S.Product_Id
```

The aggregation above produces a two-dimensional view of the data.. .why?

- The output of the previous query

Product_Id	*Market_Id*	Sum(Amount)
P1	M1	2503
P1	M2	2502
P2	6003	4566
P3	M1	2503
P3	M2	2500
P3	M3	7503
P4	…	…

Quiz…

What's the output of the following query?

```
SELECT      S.Market_Id, S.Product_Id, SUM (S.Sales_Amt)
FROM        Sales  S;
```

Quiz…

What's the output of the following query?

```
SELECT     S.Market_Id, S.Product_Id, SUM (S.Sales_Amt)
FROM         Sales  S;
```

Error: ORA-00937: not a single-group group function
00937. 00000 - "not a single-group group function"

A SELECT list cannot include both a group function, such as AVG, COUNT, MAX, MIN, SUM, STDDEV, or VARIANCE, and an individual column expression, unless the individual column expression is included in a GROUP BY clause.

Quiz…

What's the output of the following query?

```
SELECT      S.Market_Id, S.Product_Id, SUM (S.Sales_Amt)
FROM        Sales  S
GROUP BY  S.Market_Id ;
```

Quiz…

What's the output of the following query?

SELECT S.*Market_Id*, S.*Product_Id*, SUM (S.*Sales_Amt*)
FROM Sales S
GROUP BY S.*Market_Id ;*

Error: ORA-00979: not a GROUP BY expression
00979. 00000 - "not a GROUP BY expression"

The SELECT list, except by the group function, must be included in a GROUP BY clause.

Quiz (one solution)…

What's the output of the following query?

SELECT S.*Market_Id*, S.*Product_Id*, SUM (S.*Sales_Amt*)
FROM Sales S
GROUP BY S.*Market_Id* , *S.Product_Id;*

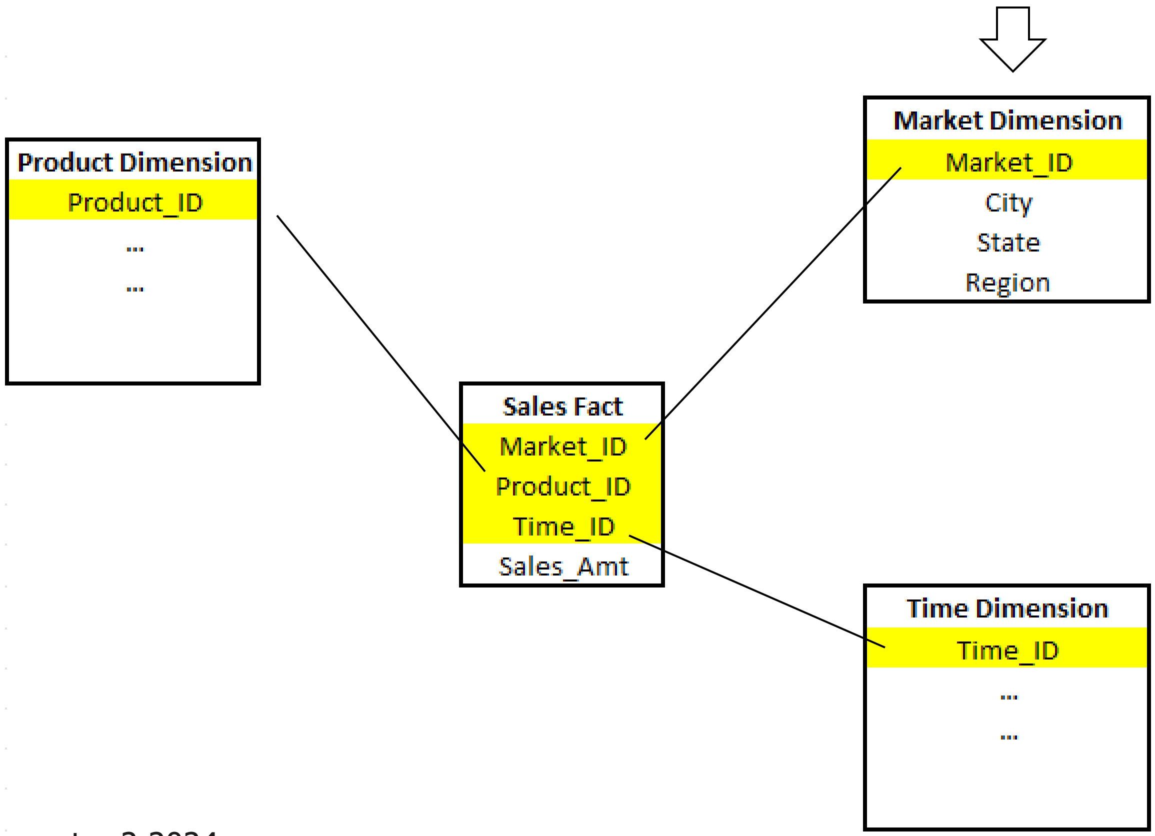
The SELECT list, except by the group function, must be included in a GROUP BY clause.

Drilling Down and Rolling Up

- Remember:

Sales (*Market_Id, Product_Id, Time_Id, Sales_Amt*)

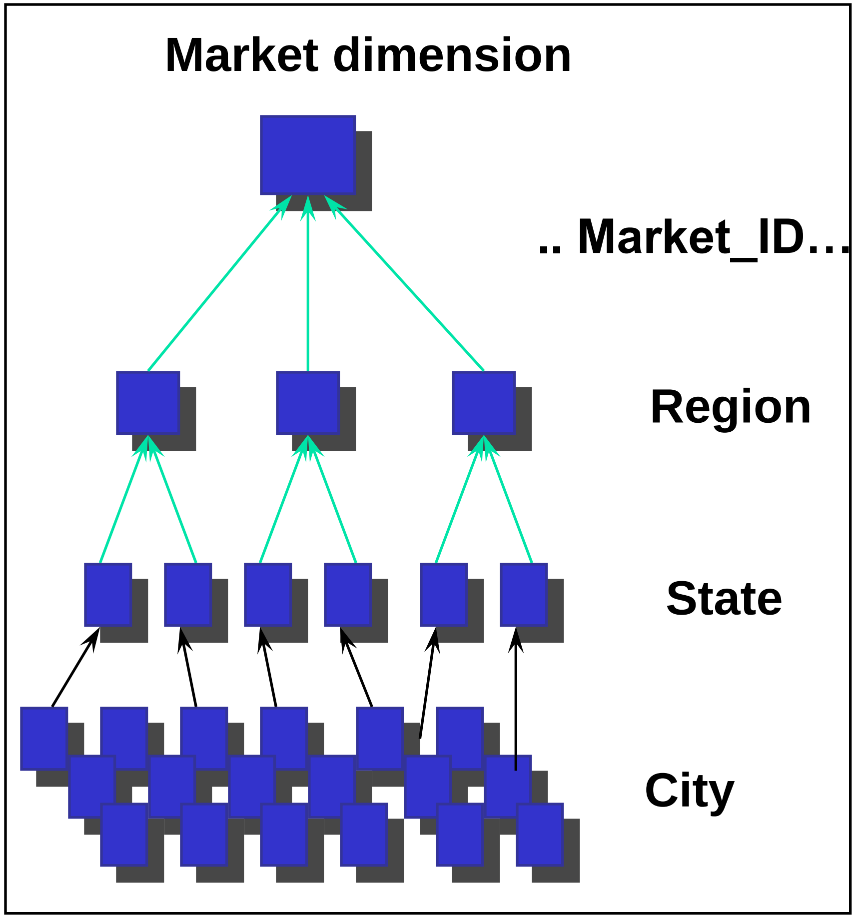
Market (*Market_Id, City, State, Region*)



Drilling Down and Rolling Up

- Some dimension tables form an ***aggregation hierarchy***

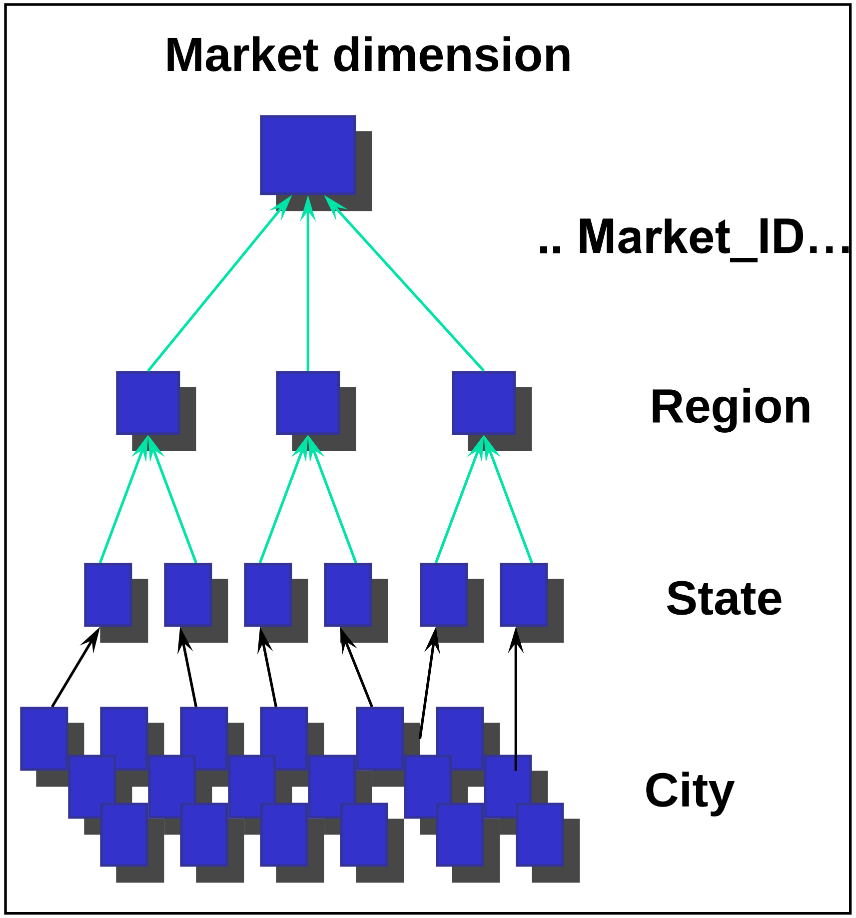
  Market_Id → City → State → Region



Drilling Down and Rolling Up

- Some dimension tables form an ***aggregation hierarchy***

  Market_Id → *City* → *State* → *Region*



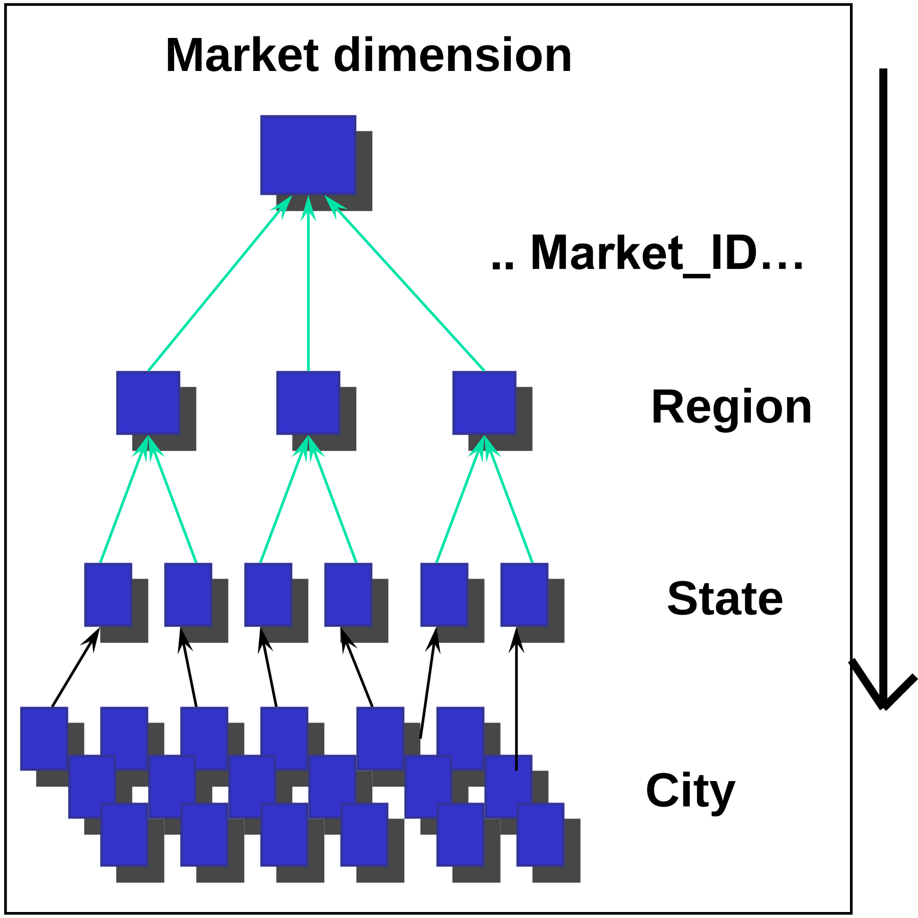
Drilling Down and Rolling Up

- Some dimension tables form an ***aggregation hierarchy***

$$Market\_Id \rightarrow City \rightarrow State \rightarrow Region$$

- Executing a series of queries that moves down a hierarchy (*e.g.*, from aggregation over regions to that over states) is called ***drilling down***

Requires the use of the fact table or information more specific than the requested aggregation (*e.g.*, cities)



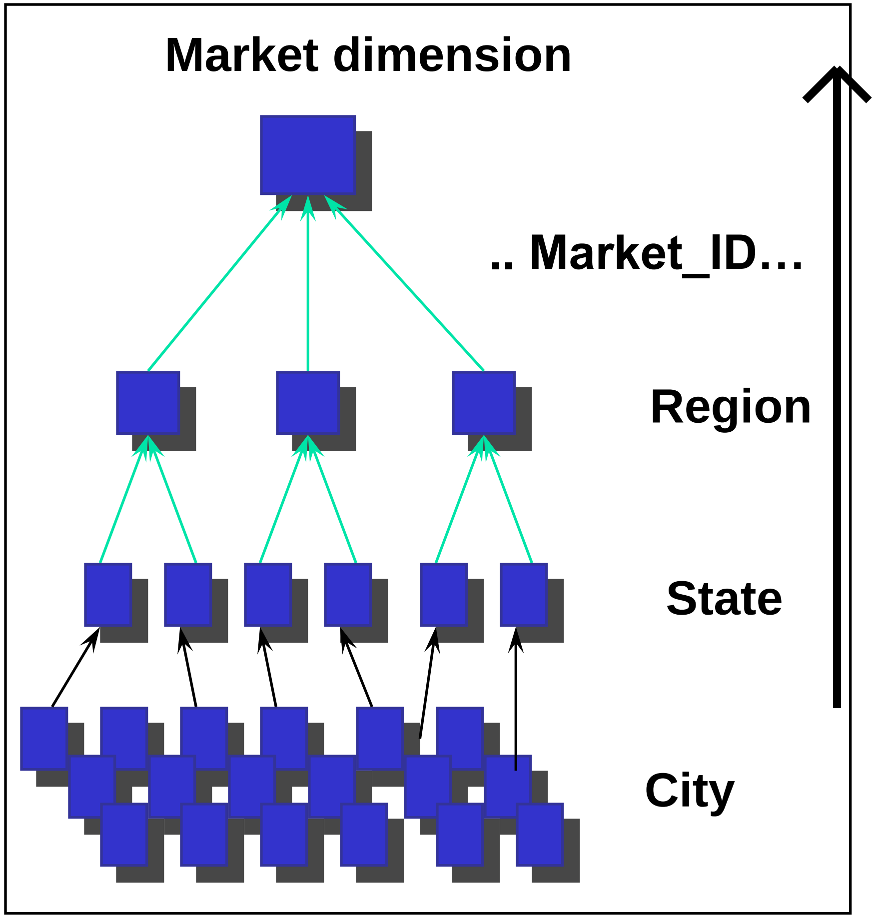
Drilling Down and Rolling Up

- Some dimension tables form an ***aggregation hierarchy***

$$Market\_Id \rightarrow City \rightarrow State \rightarrow Region$$

- Executing a series of queries that moves up the hierarchy (e.g., from states to regions) is called ***rolling up***

  HINT: In a rollup, coarser aggregations can be computed using prior queries for finer aggregations.



Drilling down

Drilling Down

- Drilling down on market: from *Region* to *State*

  Sales (*Market_Id, Product_Id, Time_Id, Sales_Amt*)

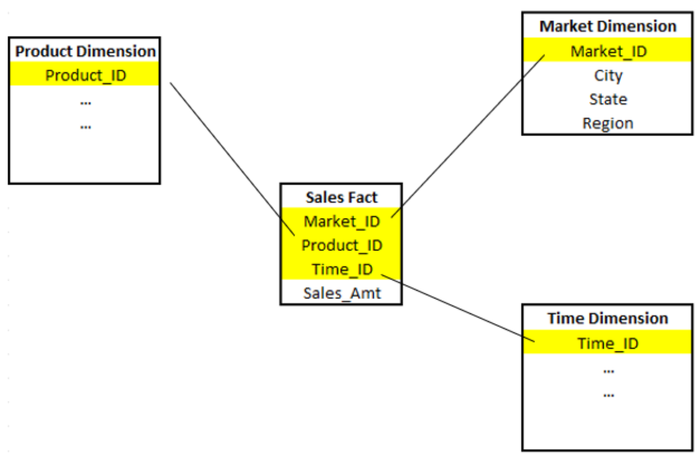
  Market (*Market_Id, City, State, Region*)

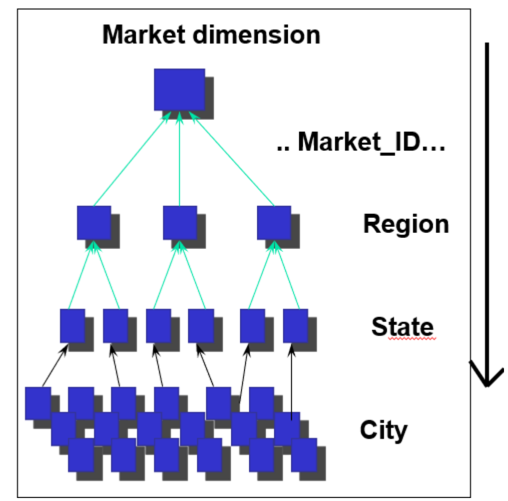
1. SELECT S.*Product_Id*, M.*Region*, SUM (S.*Sales_Amt*)

   FROM Sales S, Market M

   WHERE M.*Market_Id* = S.*Market_Id*

   GROUP BY S.*Product_Id*, M.*Region*





Drilling Down

- Drilling down on market: from *Region* to *State*

Sales (*Market_Id, Product_Id, Time_Id, Sales_Amt*)
Market (*Market_Id, City, State, Region*)

1. SELECT S.*Product_Id*, M.*Region*, SUM (S.*Sales_Amt*)
   FROM Sales S, Market M
   WHERE M.*Market_Id* = S.*Market_Id*
   GROUP BY S.*Product_Id*, M.*Region*

2. SELECT S.*Product_Id*, M.*State*, SUM (S.*Sales_Amt*)
   FROM Sales S, Market M
   WHERE M.*Market_Id* = S.*Market_Id*
   GROUP BY S.*Product_Id*, M.*State*,

Rolling Up

- Rolling up on market, from *State* to *Region*
  - If we have already created a table, State_Sales, using

1.
```
SELECT     S.Product_Id,  M.State, SUM (S.Sales_Amt)
FROM       Sales  S,   Market  M
WHERE      M.Market_Id = S.Market_Id
GROUP BY   S.Product_Id,  M.State
```

Sales (*Market_Id, Product_Id, Time_Id, Sales_Amt*)
Market (*Market_Id, City, State, Region*)

Rolling Up

- Rolling up on market, from *State* to *Region*
  - If we have already created a table, State_Sales, using

1. SELECT S.*Product_Id*, M.*State*, SUM (S.*Sales_Amt*)
   FROM Sales S, Market M
   WHERE M.*Market_Id* = S.*Market_Id*
   GROUP BY S.*Product_Id*, M.*State*

then we can roll up from there to:

2. SELECT T.*Product_Id*, M.*Region*, SUM (T.*Sales_Amt*)
   FROM State_Sales T, Market M
   WHERE M.*State* = T.*State*
   GROUP BY T.*Product_Id*, M.*Region*

Group by vs Nested Queries

1. Group by vs Nested Queries
2. Having Clause
3. More complex examples SQL querying

References:

(a) A Conceptual Poverty Mapping Data Model
Link: https://www.researchgate.net/figure/Key-thematic-layers-for-poverty-spatial-data-modeling_fig2_229724703

(b) Relational Database relationships
https://www.youtube.com/watch?v=C3icLzBtg8I

(c) https://courses.ischool.berkeley.edu/i202/f97/Lecture13/DatabaseDesign/sld002.htm

(d) https://nexwebsites.com/database/database-management-systems/

(e) Acknowledgement – Thanks to http://courses.cs.washington.edu/courses/cse544/ for providing part of this presentation.

(f) Acknowledgement – Thanks to © Silberchatz, Korth and Surdashan for providing part of this presentation.

(e) Malinowski, Elzbieta, Zimányi, Esteban (2008) *Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications .* Springer Berlin Heidelberg. Copyright © 2008 Elzbieta Malinowski & Esteban Zimányi