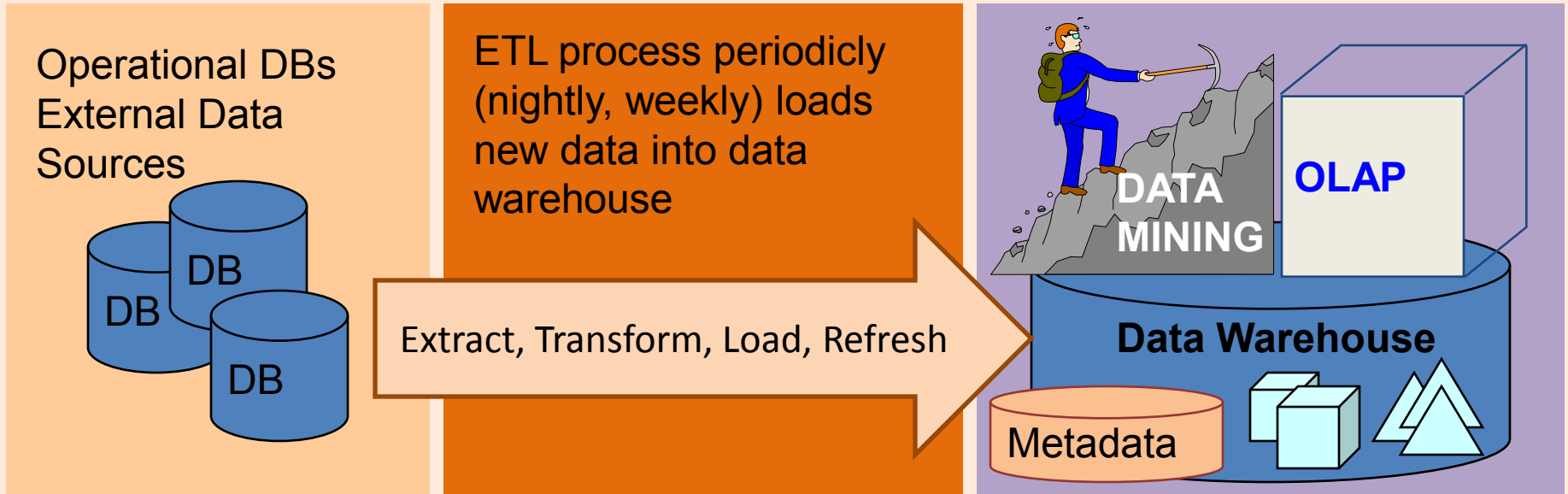


ICS 421 Spring 2010  
**Data Warehousing 2**

Asst. Prof. Lipyeow Lim  
Information & Computer Science Department  
University of Hawaii at Manoa

# Data Warehousing



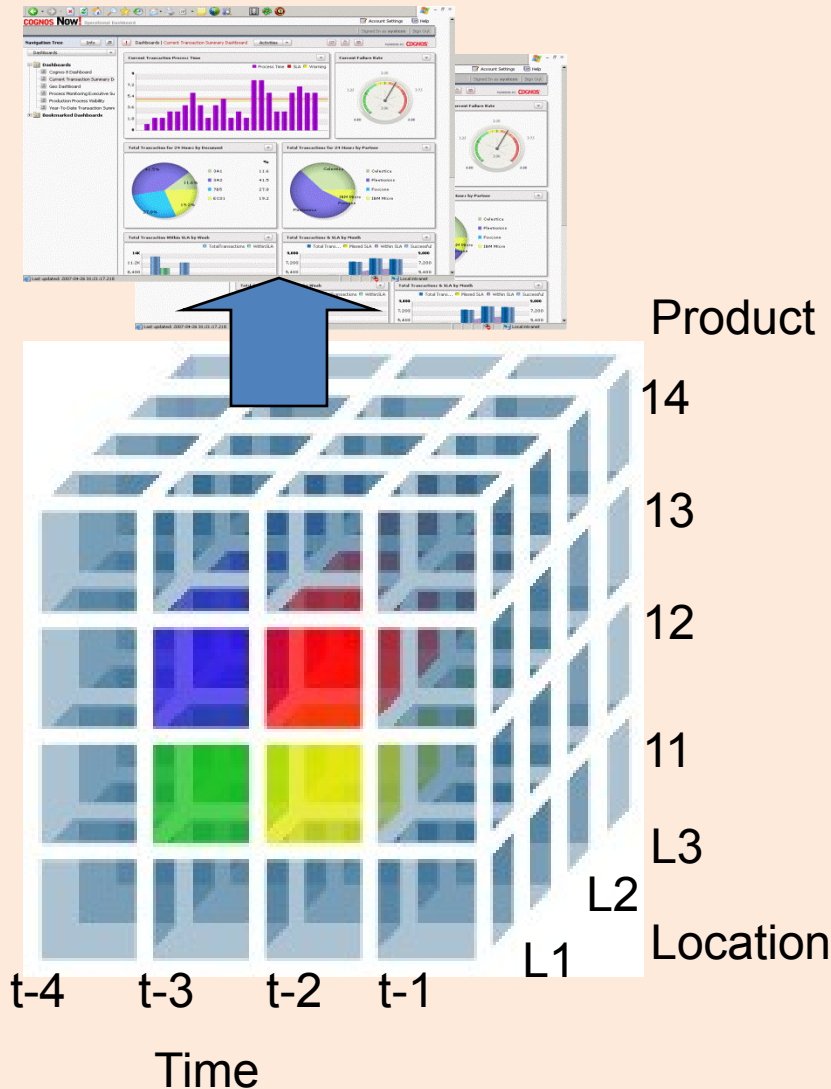
- Integrated data spanning long time periods, often augmented with summary information.
- Several terabytes common.
- Interactive response times expected for complex queries; ad-hoc updates uncommon.

# Warehousing Issues

- **Semantic Integration:** When getting data from multiple sources, must eliminate mismatches, e.g., different currencies, schemas.
- **Heterogeneous Sources:** Must access data from a variety of source formats and repositories.
  - Replication capabilities can be exploited here.
- **Load, Refresh, Purge:** Must load data, periodically refresh it, and purge too-old data.
- **Metadata Management:** Must keep track of source, loading time, and other information for all data in the warehouse.

# Multidimensional Data Model

- Collection of numeric measures, which depend on a set of dimensions.
  - E.g., measure **Sales**, dimensions **Product** (key: pid), **Location** (locid), and **Time** (timeid).

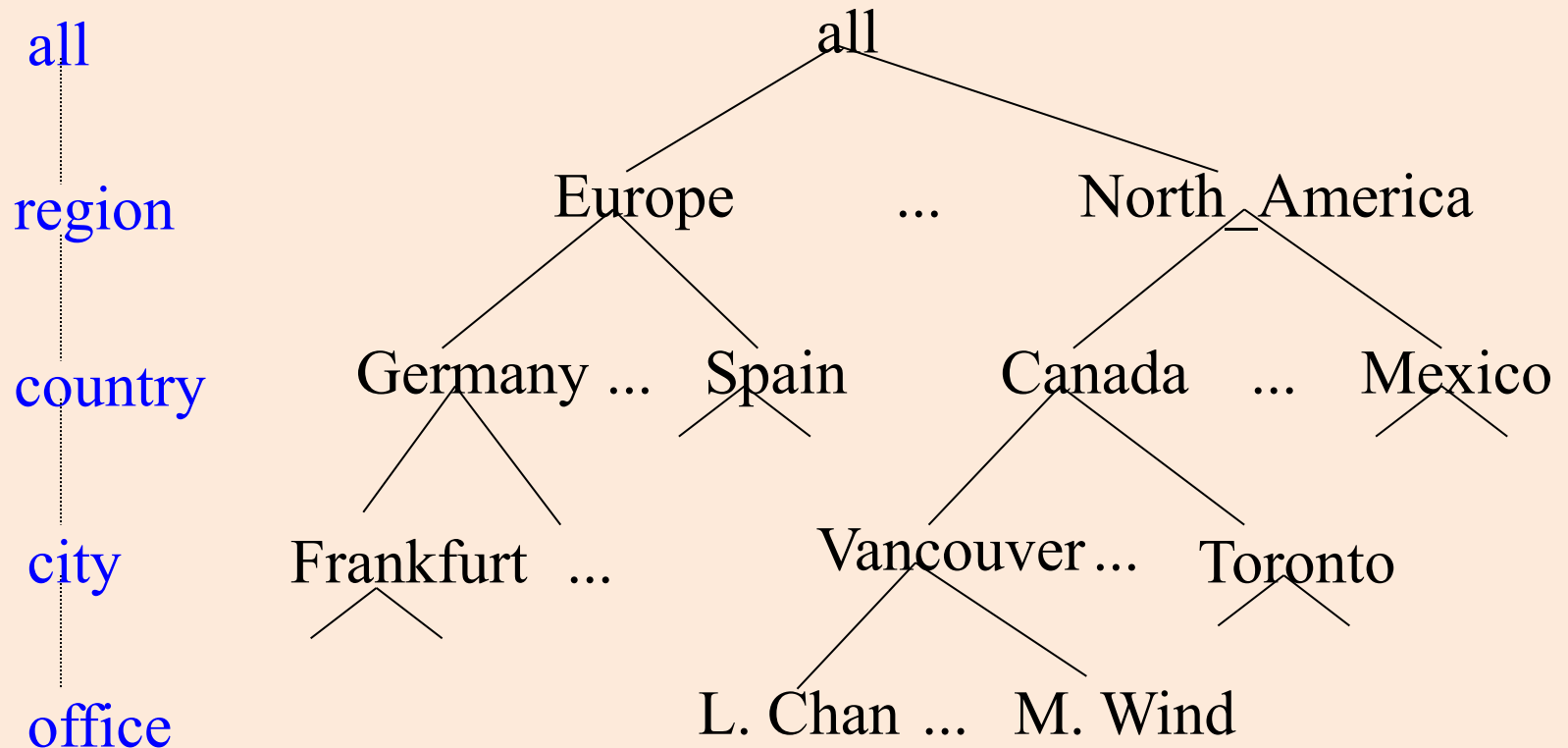


# MOLAP vs ROLAP

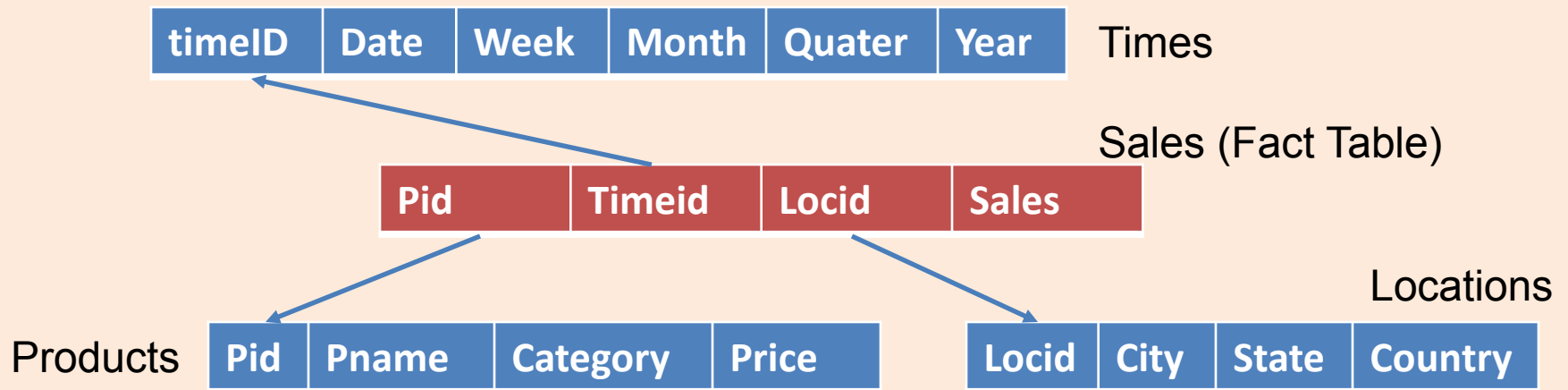
- Multidimensional data can be stored physically in a (disk-resident, persistent) array; called **MOLAP** systems. Alternatively, can store as a relation; called **ROLAP** systems.
- The main relation, which relates dimensions to a measure, is called the **fact table**. Each dimension can have additional attributes and an associated **dimension table**.
  - E.g., **Products(pid, pname, category, price)**
  - Fact tables are *much* larger than dimensional tables.

# Dimension Hierarchies

- For each dimension, the set of values can be organized in a hierarchy:

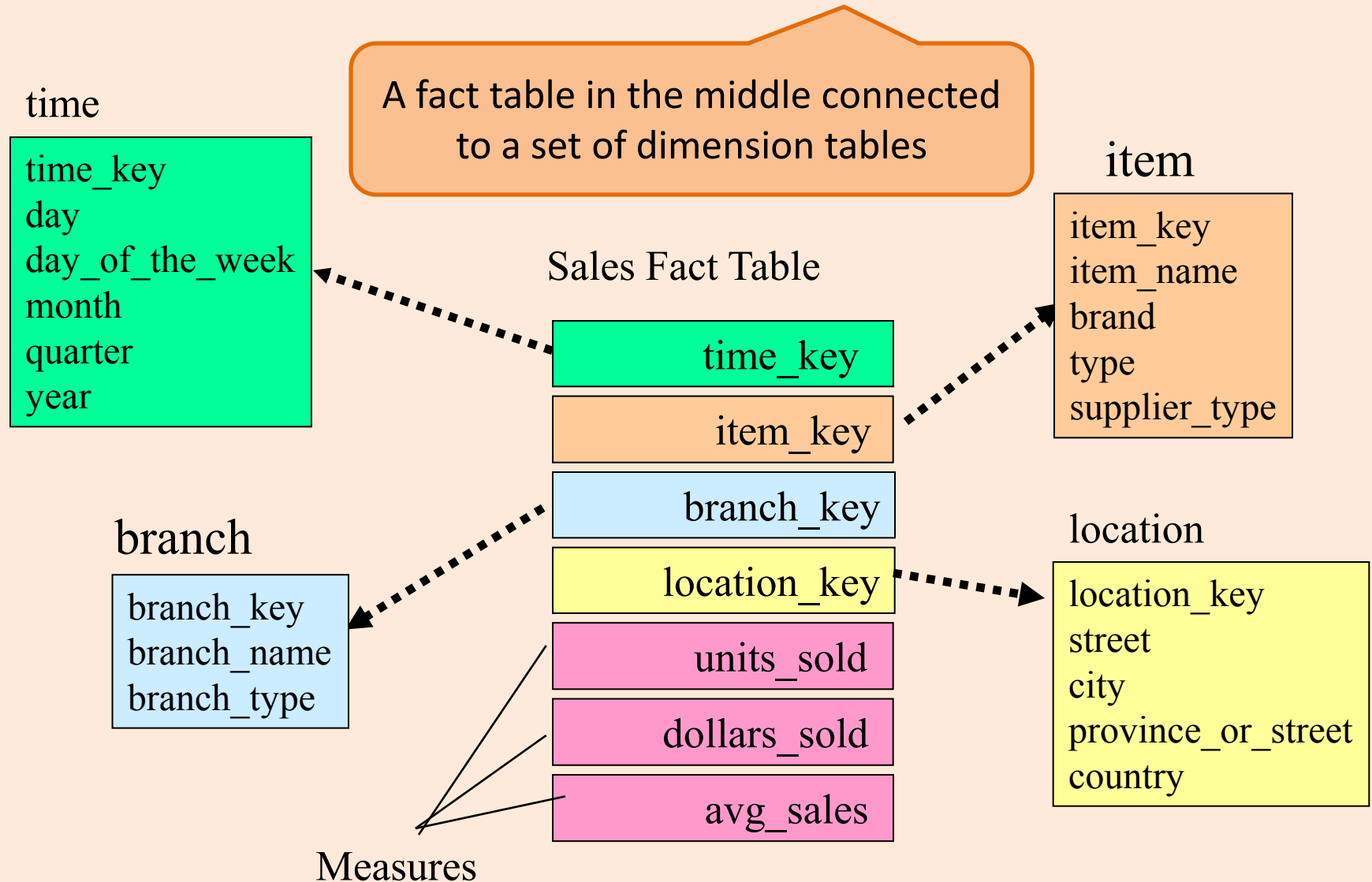


# Conceptual Design of Data Warehouses



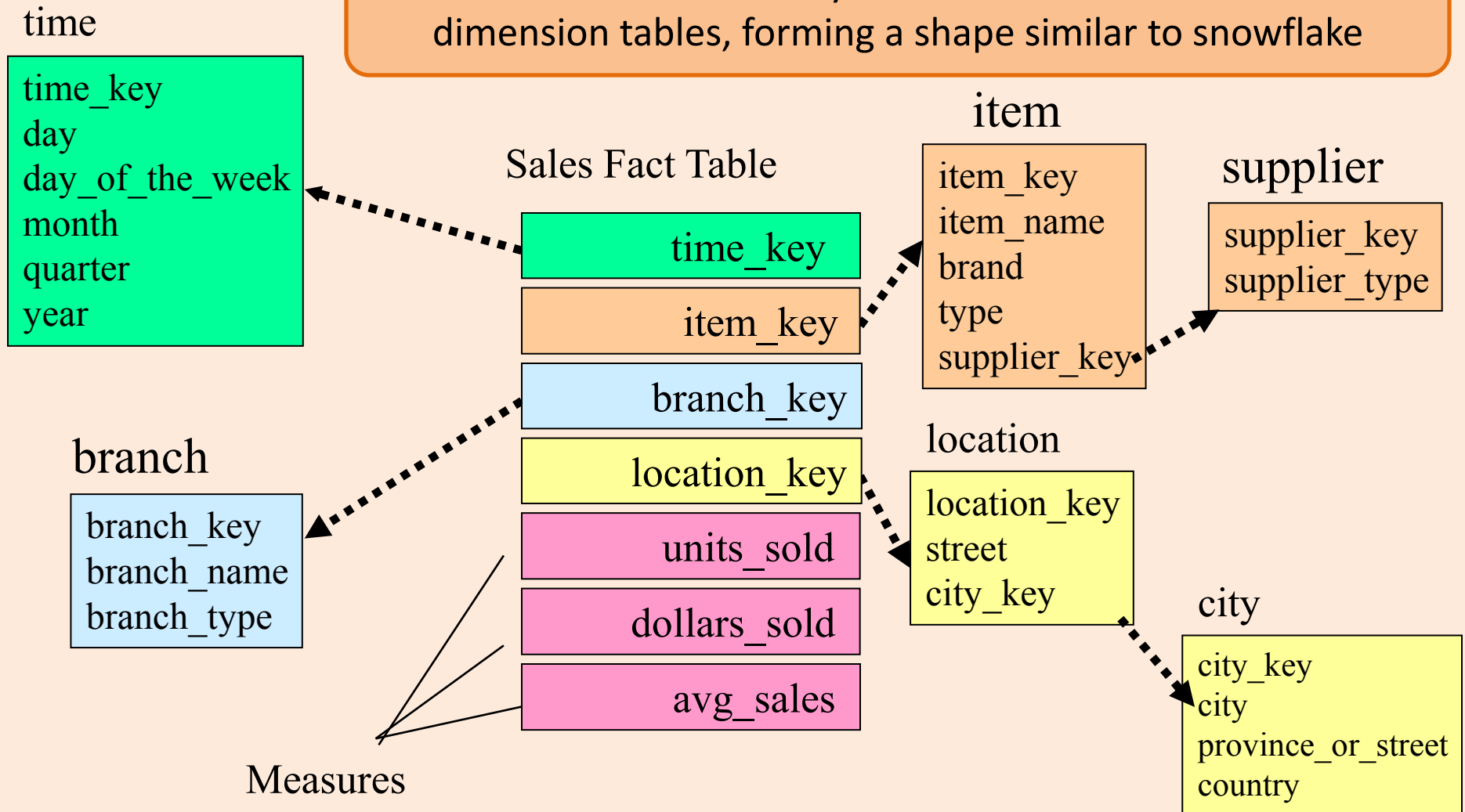
- Fact table in BCNF; dimension tables un-normalized.
  - Dimension tables are small; updates/inserts/deletes are rare. So, anomalies less important than query performance.
- This kind of schema is very common in OLAP applications, and is called a **star schema**; computing the join of all these relations is called a **star join**.

# Example: Star Schema



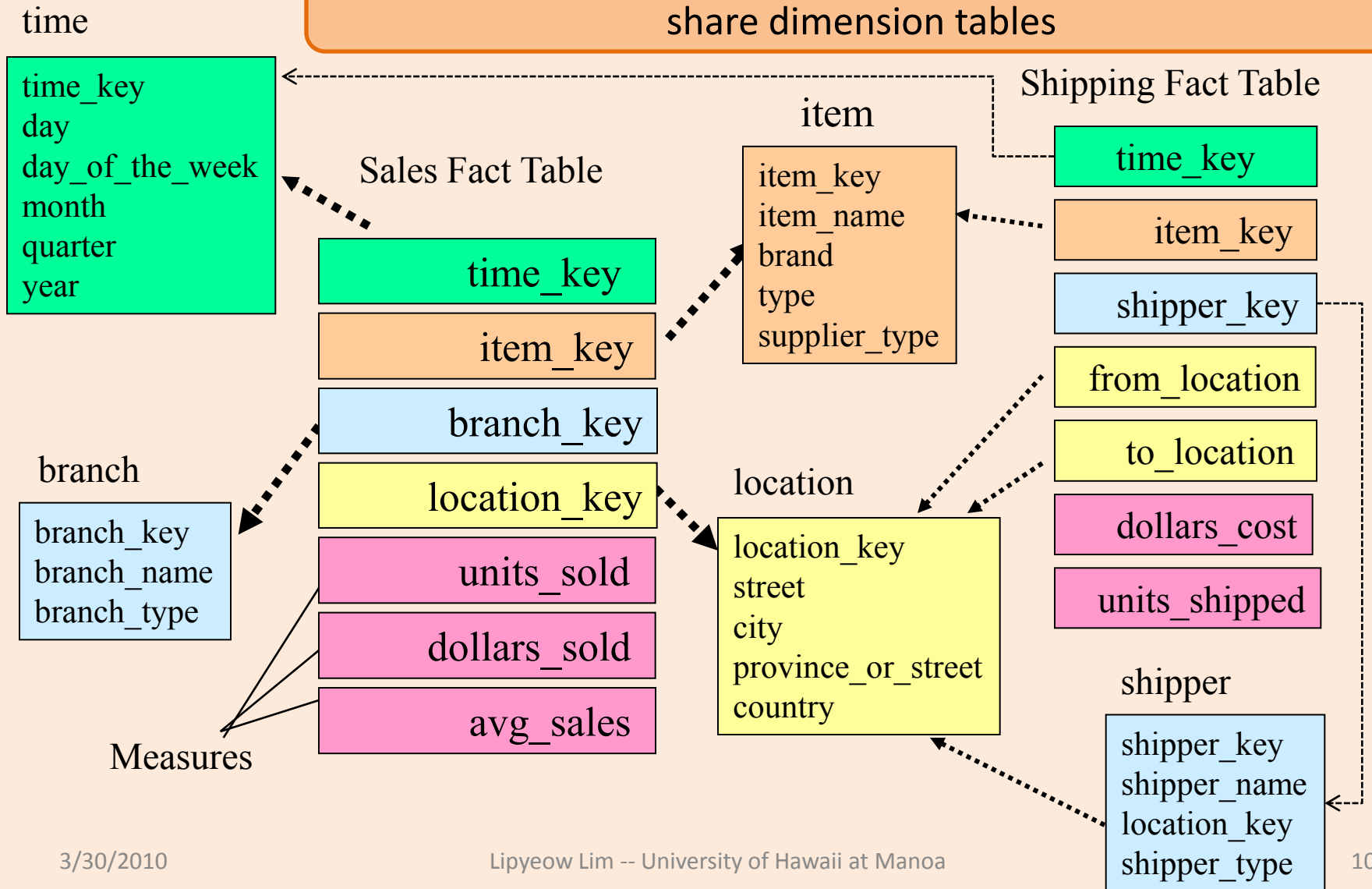
# Example: Snowflake Schema

some dimensional hierarchy is normalized into a set of smaller dimension tables, forming a shape similar to snowflake



# Example: Constellation

galaxy schema or fact constellation : Multiple fact tables share dimension tables

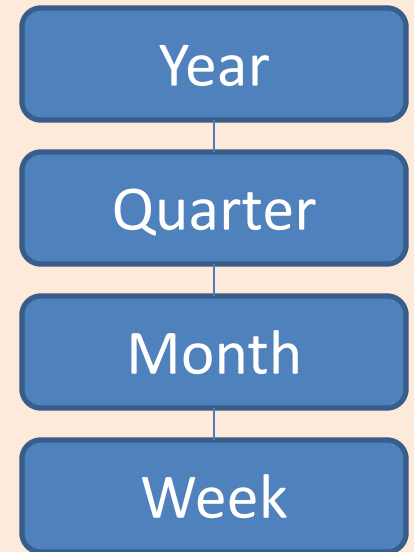


# OLAP Queries

- Influenced by SQL and by spreadsheets.
- A common operation is to aggregate a measure over one or more dimensions.
  - Find total sales.
  - Find total sales for each city, or for each state.
  - Find top five products ranked by total sales.
- Roll-up: Aggregating at different levels of a dimension hierarchy.
  - E.g., Given total sales by city, we can roll-up to get sales by state.

# More OLAP Queries

- Drill-down: The inverse of roll-up.
  - E.g., Given total sales by state, can drill-down to get total sales by city.
  - E.g., Can also drill-down on different dimension to get total sales by product for each state.
- Pivoting: Aggregation on selected dimensions.
  - E.g., Pivoting on Location and Time yields this cross-tabulation:
- Slicing and Dicing: Equality and range selections on one or more dimensions.



Year\ State	WI	CA	Total
1995	63	81	144
1996	38	107	145
1997	75	35	110
Total	176	223	339

# Comparison with SQL Queries

- The cross-tabulation obtained by pivoting can also be computed using a collection of SQLqueries:

Year\ State	WI	CA	Total
1995	63	81	144
1996	38	107	145
1997	75	35	110
Total	176	223	339

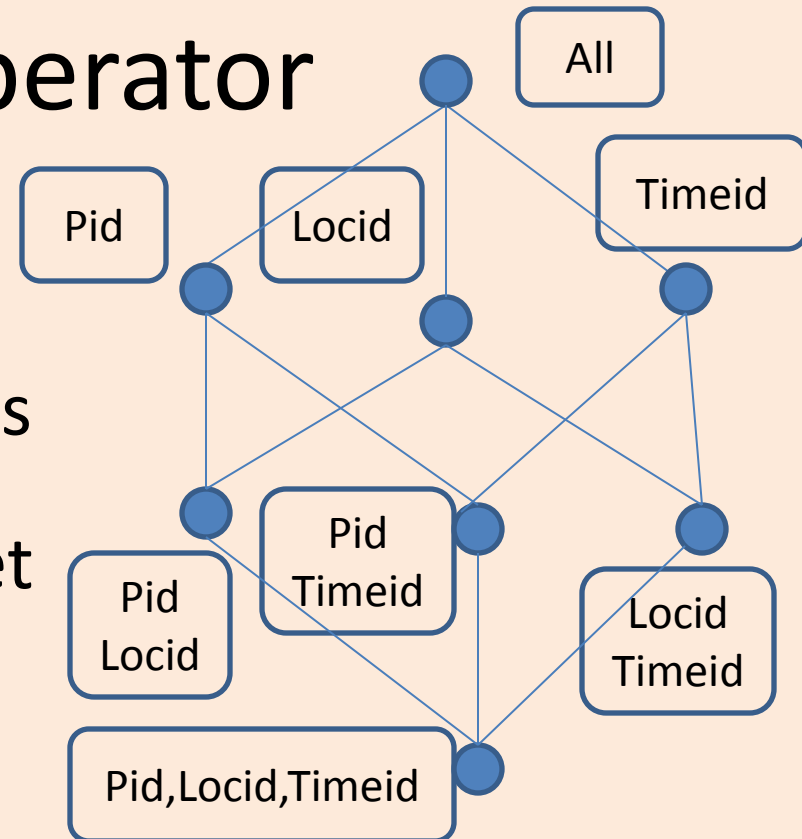
```
SELECT SUM(S.sales)
FROM Sales S, Times T, Locations L
WHERE S.timeid=T.timeid AND S.locid=L.locid
GROUP BY T.year, L.state
```

```
SELECT SUM(S.sales)
FROM Sales S, Times T
WHERE S.timeid=T.timeid
GROUP BY T.year
```

```
SELECT SUM(S.sales)
FROM Sales S, Location L
WHERE S.locid=L.locid
GROUP BY L.state
```

# The CUBE Operator

- Generalizing the previous example, if there are  $k$  dimensions, we have  $2^k$  possible SQL GROUP BY queries that can be generated through pivoting on a subset of dimensions.
- CUBE pid, locid, timeid BY SUM Sales
  - Equivalent to rolling up Sales on all eight subsets of the set {pid, locid, timeid}; each roll-up corresponds to an SQL query of the form:



```
SELECT SUM(S.sales)
FROM Sales S
GROUP BY grouping-list
```

Lots of work on optimizing the CUBE operator

# Querying Sequences in SQL:1999

- Trend analysis is difficult to do in SQL-92:
  - Find the % change in monthly sales
  - Find the top 5 product by total sales
  - Find the trailing  $n$ -day moving average of sales
  - The first two queries can be expressed with difficulty, but the third cannot even be expressed in SQL-92 if  $n$  is a parameter of the query.
- The WINDOW clause in SQL:1999 allows us to write such queries over a table viewed as a sequence (implicitly, based on user-specified sort keys)

# The WINDOW Clause

```
SELECT L.state, T.month, AVG(S.sales) OVER W AS movavg
FROM Sales S, Times T, Locations L
WHERE S.timeid=T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.state
ORDER BY T.month
RANGE BETWEEN INTERVAL `1` MONTH PRECEDING
AND INTERVAL `1` MONTH FOLLOWING)
```

- Let the result of the FROM and WHERE clauses be “Temp”.
- (Conceptually) Temp is partitioned according to the PARTITION BY clause.
  - Similar to GROUP BY, but the answer has one row for each row in a partition, not one row per partition!
- Each partition is sorted according to the ORDER BY clause.
- For each row in a partition, the WINDOW clause creates a “window” of nearby (preceding or succeeding) tuples.
  - Can be value-based, as in example, using RANGE
  - Can be based on number of rows to include in the window, using ROWS clause
- The aggregate function is evaluated for each row in the partition using the corresponding window.
  - New aggregate functions that are useful with windowing include RANK (position of a row within its partition) and its variants DENSE\_RANK, PERCENT\_RANK, CUME\_DIST.

# Top K Queries

- If you want to find the 10 (or so) cheapest cars, it would be nice if the DB could avoid computing the costs of all cars before sorting to determine the 10 cheapest.
  - **Idea:** Guess at a cost  $c$  such that the 10 cheapest all cost less than  $c$ , and that not too many more cost less. Then add the selection  $\text{cost} < c$  and evaluate the query.
    - If the guess is right, great, we avoid computation for cars that cost more than  $c$ .
    - If the guess is wrong, need to reset the selection and recompute the original query.

# Example: Top K Queries

```
SELECT P.pid, P.pname, S.sales  
FROM Sales S, Products P  
WHERE S.pid=P.pid AND S.locid=1 AND S.timeid=3  
ORDER BY S.sales DESC  
FETCH FIRST 10 ROWS ONLY
```

```
SELECT P.pid, P.pname, S.sales  
FROM Sales S, Products P  
WHERE S.pid=P.pid AND S.locid=1 AND S.timeid=3  
AND S.sales > c  
ORDER BY S.sales DESC
```

- **FETCH FIRST 10 ROWS ONLY** is not in SQL99
- Cut-off value  $c$  is chosen by optimizer

# Online Aggregation

- Consider an aggregate query, e.g., finding the average sales by state. Can we provide the user with some information before the exact average is computed for all states?
  - Can show the current “running average” for each state as the computation proceeds.
  - Even better, if we use statistical techniques and sample tuples to aggregate instead of simply scanning the aggregated table, we can provide bounds such as “the average for Wisconsin is  $2000 \pm 102$  with 95% probability.”
    - Should also use nonblocking algorithms!