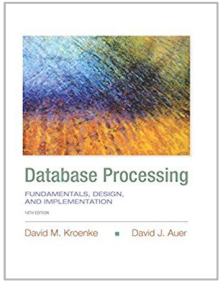
Fairfield University
DOLAN
School of Business




# Databases for Analytics

Kroenke / Auer
Chapter 12

# Learning Objectives

- **Skills:** You should know how to ...
  - Use star and snowflake design schema to architect a dimensional data warehouse
- **Theory:** You should be able to explain ...
  - The implications of "big data" for analytics
  - The difference between transaction processing and analytical processing
  - The motivation and essential features of common NoSQL database designs

# Big Data

And its implications for data analysts

# Data storage capacity timeline

- Pre-1980: Lab coats, punch cards and magnetic tape
  - Each card could record ~80 ***bytes*** of data
- 1980-1990: Rise of the PC
  - Floppy: 8" (80 ***KB***) → 5 ¼ " (110 KB) → 3 ½ " (320 KB)
  - Hard Drive: IBM XT (10 ***MB***) → Mac IIX (80-100 MB)
- 1990-2000 Birth of E-Commerce
  - Mainframes replaced by web-fronted DBMSes with ~5 GB or more capacity → Slashdot Effect → Cloud Computing
- 2000-Now Scaling Up
  - Google has about 15 ***exabytes*** ($15 \times 10^{18}$ bytes) of data

# Implications for Data Analysts

- Pre 1980
  - "Any amount of **programming and fancy math** to avoid collecting more data or buying more hardware"
- 1980-2000
  - "Any amount of **hardware** to avoid leaking data or buying more software licenses"
- 2000-Now
  - "Any amount of **smarts** to wring useful insights from commodity hardware, software, ***and data***"

# How big is big? (And does size matter?)

| Name | Symbol | Approximate Value for Reference | Actual Value |
|---|---|---|---|
| Byte | | | 8 bits [Store one character] |
| Kilobyte | KB | About $10^3$ | $2^{10}$ = 1,024 bytes |
| Megabyte | MB | About $10^6$ | $2^{20}$ = 1,024 KB |
| Gigabyte | GB | About $10^9$ | $2^{30}$ = 1,024 MB |
| Terabyte | TB | About $10^{12}$ | $2^{40}$ = 1,024 GB |
| Petabyte | PB | About $10^{15}$ | $2^{50}$ = 1,024 TB |
| Exabyte | EB | About $10^{18}$ | $2^{60}$ = 1,024 PB |
| Zettabyte | ZB | About $10^{21}$ | $2^{70}$ = 1,024 EB |
| Yottabyte | YB | About $10^{24}$ | $2^{80}$ = 1,024 ZB |



Big Data is actually a misnomer of sorts.

While it is easy to classify data stores based on their physical size, "big data" is really about a set of operating characteristics.

# Big (and not so big) Data Characteristics

- Dirty Data/Missing Values: bad domain integrity
- Inconsistent Data: differing sources and/or times affect the way data is collected (entity integrity)
- Nonintegrated Data: multiple data silos with no / imperfect common keys (referential integrity)
- Ill-structured Data: NoSQL anyone? Twitter feeds?
- Too Much Data
  - Too many data columns: normalization problems
  - Too many data rows: performance issues

# Big Data Analytics is ... Pig Ugly

In the end, **data is just data**. We can expect "big data" problems even with small data sets.

However, ***every* weird data problem** you can think of is **virtually guaranteed** to exist in massively big data sets. It's like fate!

***If it's big, assume it's ugly. Expect to spend 80% of your time applying lots of lipstick.***

# Business Intelligence

Using Data to Make Smart Decisions

# Online vs Analytical Processing

**Online (operational) systems** capture data at the source

- Generally designed around **transactions** that generate data
- Data is constantly changing to reflect current status

**Analytical (BI) systems** support decisions or provide insights

- Generally designed around the analytical requirements
- Offline data, only as timely as needed by the requirements

# Two Essential BI Functions

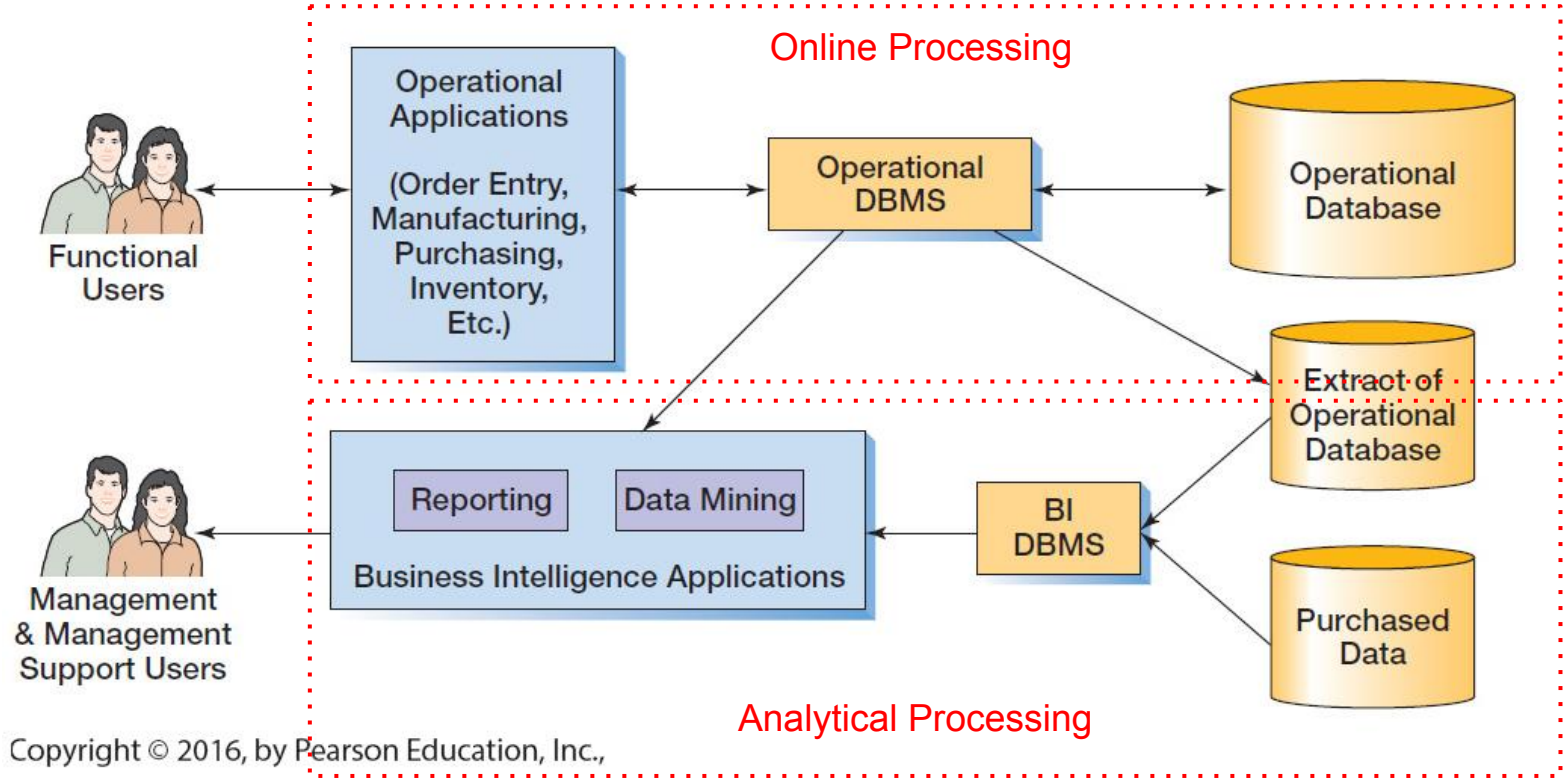
**Reporting** (BA 540)

- Sort, filter, group, and make rudimentary calculations based on available data
- Focus is on decisions / tasks that may occur repeatedly

**Data Mining** (BS 545)

- Sophisticated analyses that generally involve complex statistical or mathematical processing
- Focus is on discovering/validating insights from data; insights may ultimately drive decisions but that's a happy side effect

# Enterprise BI Architecture



# Data Warehouses

Where Structured Analytical Data
Chooses to Live

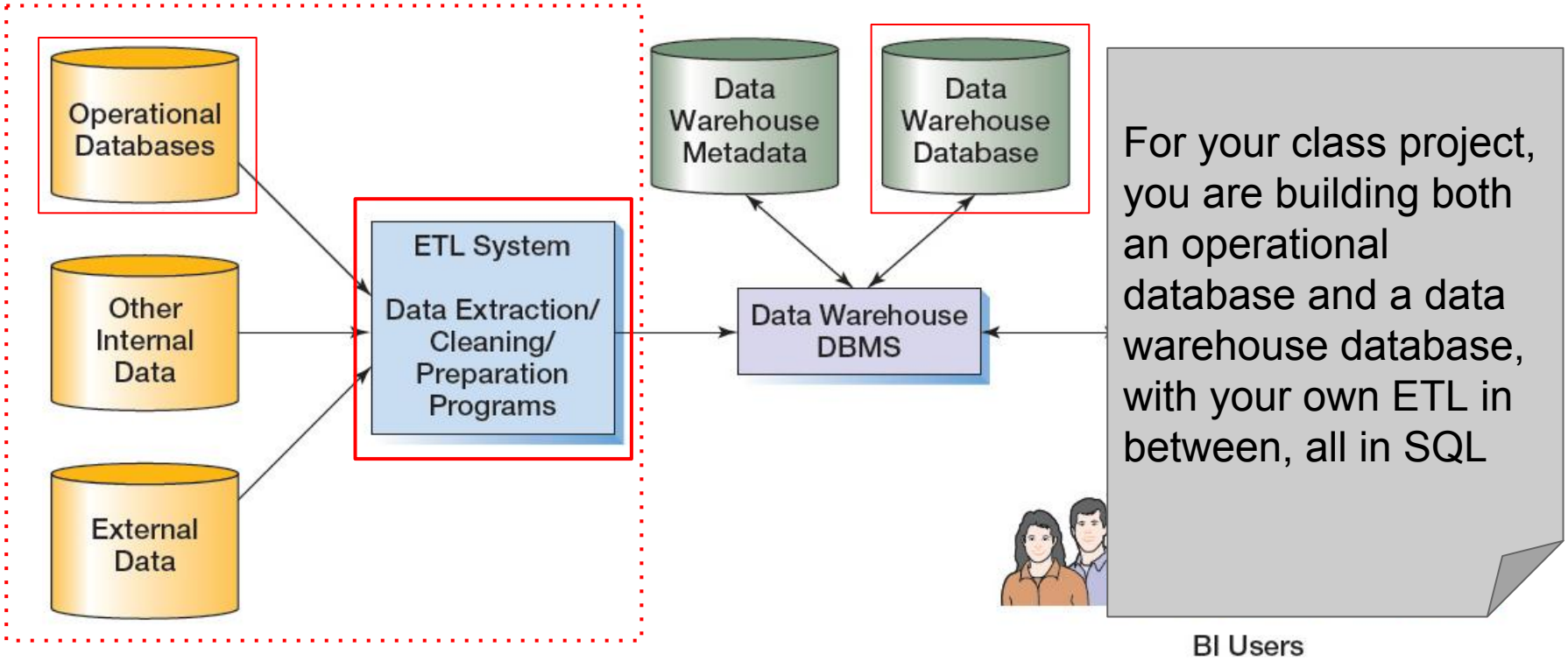
# Why Not Just Use Operational Data?

- Performance & Complexity
  - Transaction systems are designed to INSERT/UPDATE one row of data at a time
  - Analytical systems tend to SELECT data from many tables in each query, which means lots of JOINS!
- Personnel & Skillsets
  - It's like the difference between a beat cop and a detective
  - Maintaining an operational system is mostly mechanical
  - Maintaining (and evolving) a data warehouse requires deep understanding of the business domain

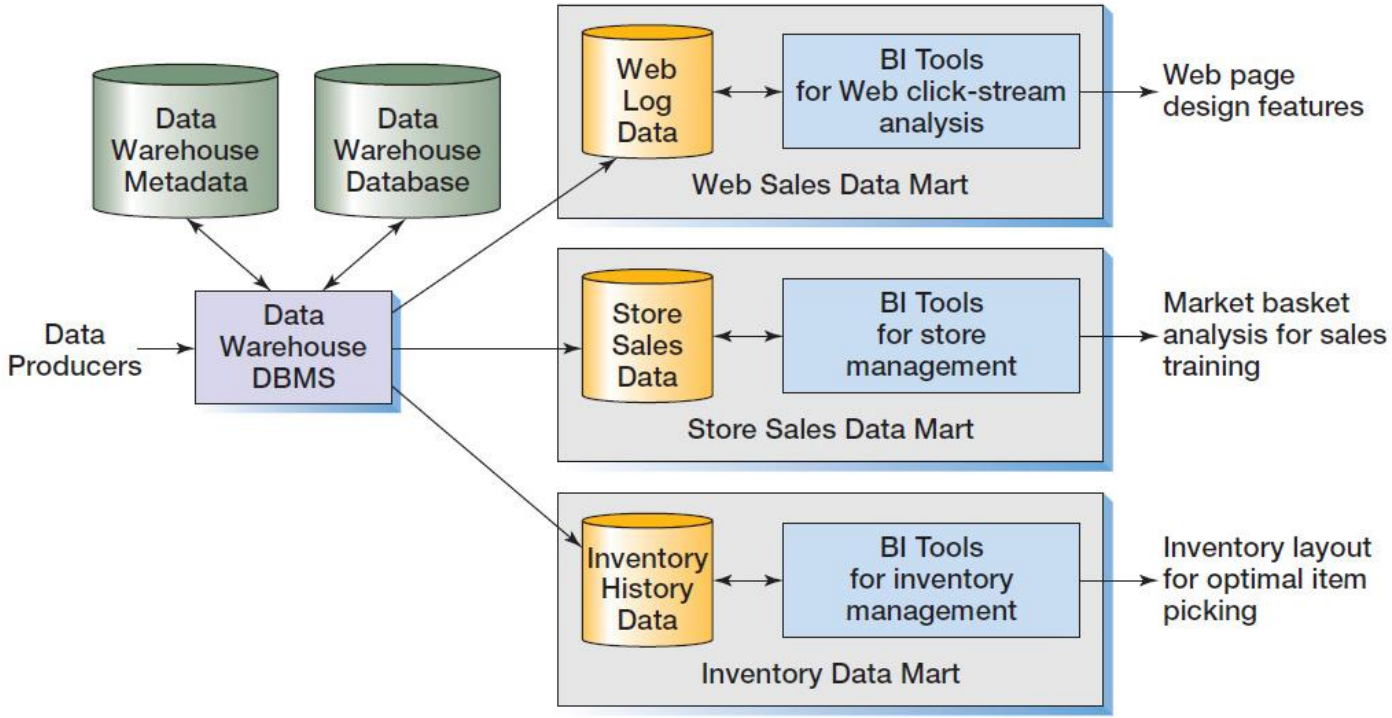
# Key Characteristics → Implications

- **DW are read-only** → we can *denormalize* the data without risking anomalies; let the operational database handle that!
- **DW are user-centered** → analysts often draw data directly from the DW, so the design needs to be tailored to their needs (questions, formats, etc.)
- **DW are often huge** → care must be taken to keep just (and all) the relevant historical data
- **DW span operational systems** → integrating data from multiple sources has special challenges

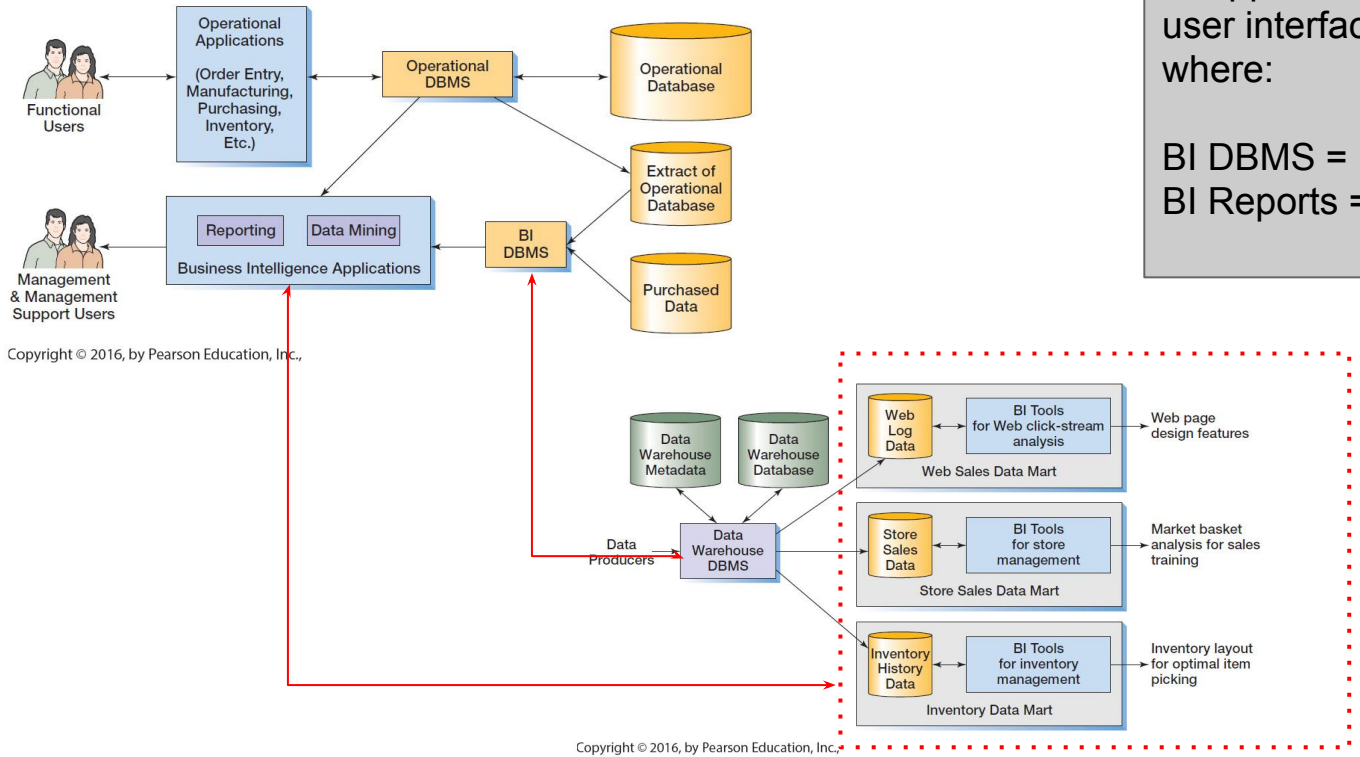
# Extract → Transform → Load



# Data Marts for Specific User Groups



# Enterprise DW Architecture




BI applications provide friendly user interfaces for the DW, where:

BI DBMS = DW DBMS
BI Reports = Data Mart Tools

# Dimensional Data Warehouses

Just enough analytical detail,
organized in the most convenient way

# Operational vs DW RDBMS (again)

| Operational Database | Dimensional Database |
| --- | --- |
| Used for structured transaction data processing | Used for unstructured analytical data processing |
| Current data are used | Current and historical data are used |
| Data are inserted, updated, and deleted by users | Data are loaded and updated systematically, not by users |

Your book cites dimensional databases here, but actually these requirements apply to all DW designs

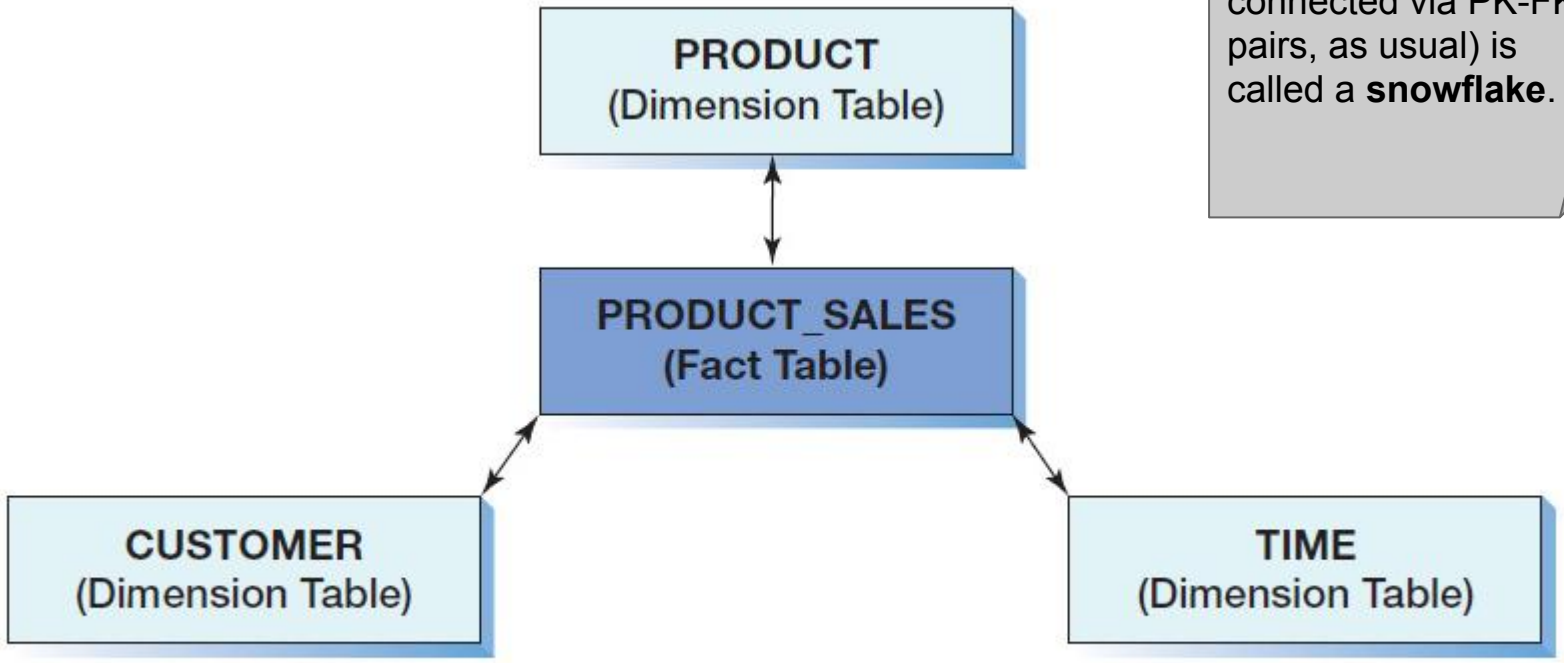
# What's a Dimensional Database?

Dimensional DBs are **relational,** with two kinds of tables:

- **Fact tables** with normalized **quantitative** data that can be aggregated as needed
  - tend to be huge, with many rows and columns
- **Dimension tables** that provide attributinal data (tags) that be used to **group** or **enrich/explain** rows in the fact tables
  - tend to be small, with just a few rows and columns

# The Star Schema

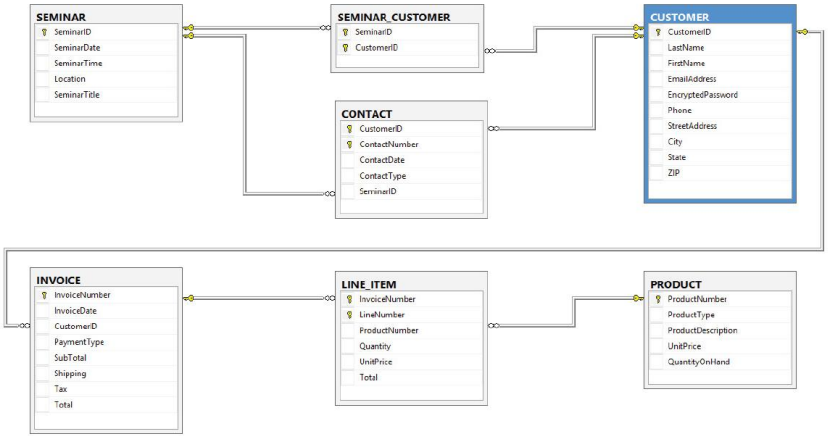
A dimensional DB with more than one fact table (which can be connected via PK-FK pairs, as usual) is called a **snowflake**.



# A Side-by-Side Comparison

We'll come back to this next time, when we talk about DW design

## Operational DB





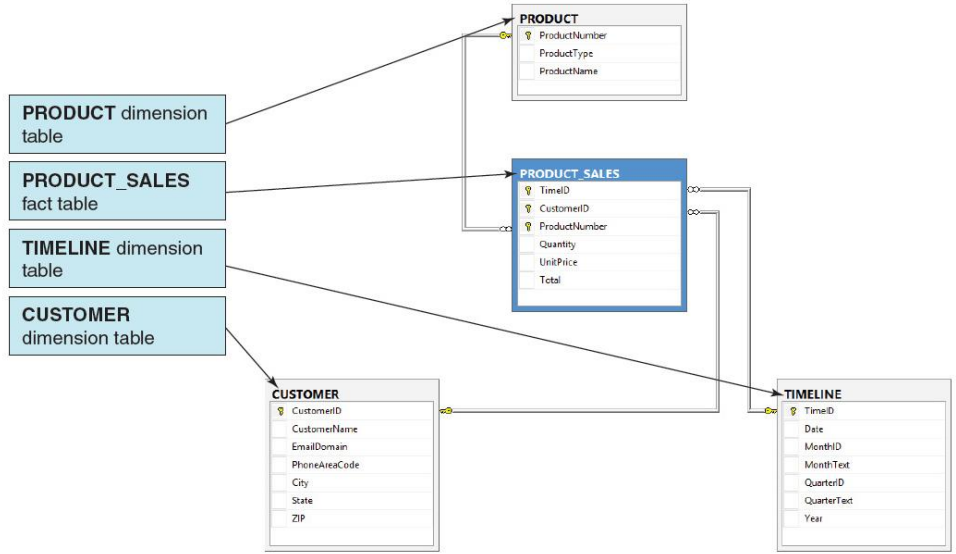

## Dimensional DB

**PRODUCT** dimension table

**PRODUCT_SALES** fact table

**TIMELINE** dimension table

**CUSTOMER** dimension table



# Ok, so what?

Ever used pivot tables?
Then you've done this!

When designed properly, a dimensional database reduces virtually all essential SQL queries to

**SELECT** *a few statistics and dimensional attributes*

**FROM** *one fact table joined with all applicable dimension tables*

**WHERE** *the facts have specified attributes or conditions*

**GROUP BY** *one or more dimensional attributes*

***Common queries can be pre-defined (including JOINS), so that the user just has to click on a few attributes (to restrict on or group by) and a few predefined statistics.***

# Roll Ups and Drill Downs

Using a SELECT query to aggregate data by groups is called a **roll up**:

Dimensional Data → Aggregated Info

A **drill down** does the opposite:

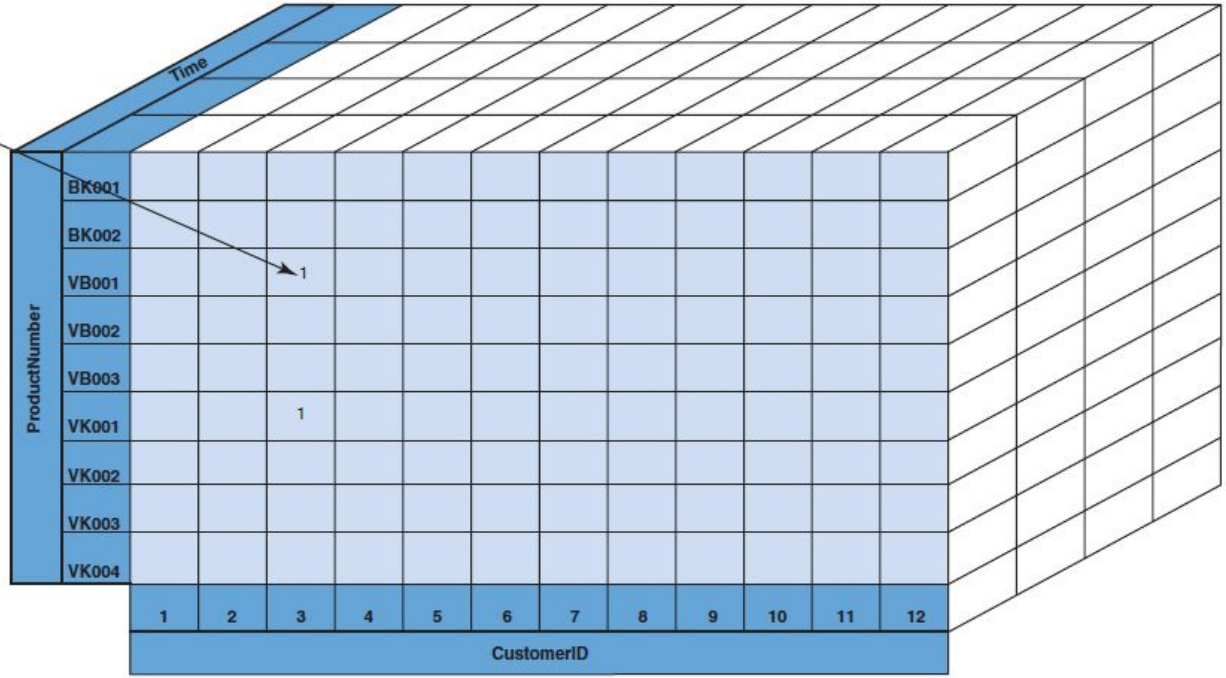
Aggregated Info → Dimensional Data

# Online Analytical Processing (OLAP)

- Origins: IBM and SAP in 1990s
- Goal: provide "near-real time" access to operational data with many of the advantages of offline DW
- Essentials
  - Automated, continuous updating of the fact tables in the data warehouse
  - Visualization tools (e.g., pivot tables in Excel) that "slice and dice" data for ad hoc analyses

# OLAP Cubes (for fast rollups & drill downs)

Each cell will show the total quantity of each product that has been purchased by each customer on a specific date



# Distributed Databases

When one database instance is not enough!

# Distributed? How? Why?

A **distributed database** is stored and processed on more than one computer.

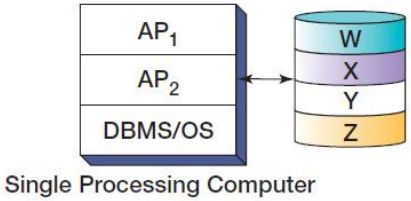
Two approaches

- **Partitioning**: data is split into separate databases
- **Replication**: data is stored and sync'ed in several (redundant) databases

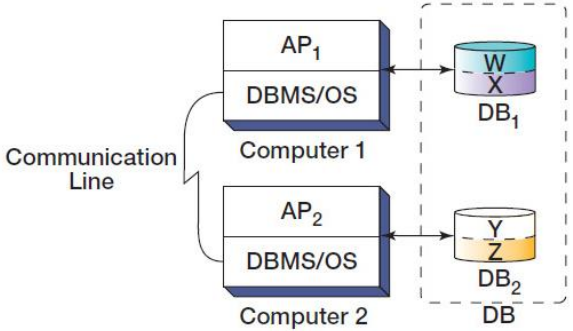
Usually, **the motivation is performance** -- keeping data physically closer to where it is used -- but there are other considerations in play

# Examples

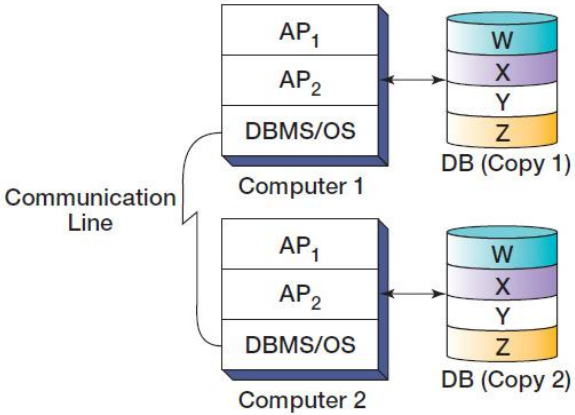
Can use both approaches



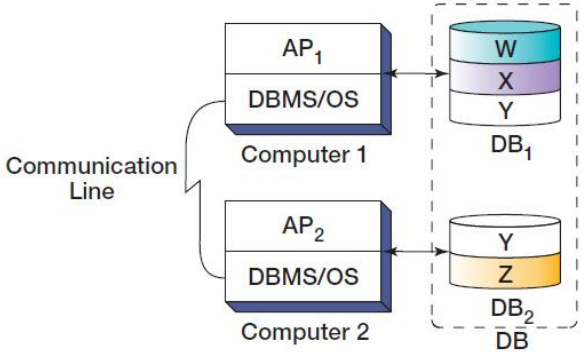

(a) Nonpartitioned, Nonreplicated Alternative




(b) Partitioned, Nonreplicated Alternative




(c) Nonpartitioned, Replicated Alternative




(d) Partitioned, Replicated Alternative

# Common Challenges

Git is a distributed data architecture. You have seen all of these things this semester!

Replication requires **synchronization** and **locking:**

- How do we keep multiple copies consistent with each other?
- If two copies are changed in conflicting ways, then which copy "wins"

Partitioning requires data **integration** (possibly over large distances)

- If we need data from DB 1 and other data from DB 2, how much latency are we comfortable with?

# Object Relational DB

Make the app programmer's job
as easy as possible

# Objects vs Entities

When used in a modern Object Oriented Programming language (like Python, Javascript, C#, Java, etc.) data takes the form of **objects** that combine:

- Properties (fields)
- Methods (operations on fields)

Not every object is an entity but every object *stored in the database* is! Confused yet?

# Goal: Object Persistence

A software object in your running code is considered ***emphemeral***

- Shut down the program → everything in memory is forgotten

An object that can be recalled from disk is considered **persistent**

- Load the program **and data** → retrieve from a persistent data store (disk)

# Object-Relational Impedance

To make programming easier, we would like to store objects directly in a relational database but there are some problems with that:

- Domain integrity: data types don't convert
- Entity integrity: objects may not be unique or even have identifiers
- Relational integrity: OOP models have a much richer set of relationships that don't exist on ERDs

# Object Persistence Technology

**Object *Oriented*** Databases store objects without using tables

- Lack of relational model has obvious integrity issues
- Idea went dead for a couple decades but has risen Zombie-like with MongoDB (more about that later)

**Object *Relational*** Databases ***map*** table rows to objects, translating from objects → tables → objects as needed

- Object relational mapping packages (e.g., SqlAlchemy) exist for just about every modern programming language.

# Virtualization and the Cloud

Making things *truly* portable and ubiquitous

# Virtualization, Hosts, and Containers

A **virtual machine** is a software implementation of a hardware system. The operating system runs inside the virtual machine like it would on physical hardware.

Every virtual machine must of course exist within a physical **host machine.**

A **container** is a stripped down virtual machine that borrows software with its host to eliminate resource duplication.

# Cloud Computing

**Cloud = Internet at commodity prices**

A virtualized container instance accessed over the internet is said to be ***running in the cloud***.

Most of the advantages attributes to cloud computing are actually due to virtualization:

- Containers can scale up or down as resources needs change
- Containers can move from one host to another without shutting down

# Our BA Lab is a Cloud App

- Host Machine: located at Amazon Web Services with 32 GB of RAM
- VM Operating System: Ubuntu Linux
- JupyterHub is a **web app** that spawns a separate JupyterLab **container** for each student
  - Technically, it's venv or Docker running JupyterLab
  - It's as if you each have your own personal JupyterLab computer

# NoSQL

Not (only) SQL? Really?

# Document DBs

- Problem addressed: Object persistence
- Solution: each object is treated as a document (like a word doc)
- Tech: MongoDB, CouchDB, etc.
- Where it fails: no set schema, so no easy way to search documents

# Key-Value Stores

Problem addressed: sparse tables with lots and lots and lots of missing values

Solution: only store the non-missing values as Entity-Attribute-Value triplets in a single table

Tech: Memcached, Redis

Where it fails: No set schema, just like document DBs

# Column Family Stores

Problem addressed: data that is organized as columns (vectors) instead of as rows (tuples)

Solution: treat tables as dictionaries of columns (like Pandas!)

Tech: Apache Cassandra

Where it fails: adding one "record" requires adding to possibly many, many columns

# Graph Stores

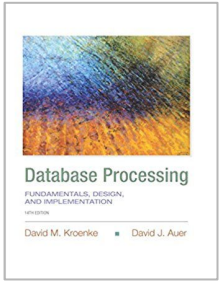
Problem addressed: spatial data that represents locations (nodes) and links (directed arcs)

Solution: Store the nodes and arcs as a single graph

Tech: Neo4j, ArcGIS

When it fails: non-spatial data

Fairfield University
DOLAN
School of Business




# Databases for Analytics

Kroenke / Auer
Chapter 12