INTRODUCTION TO DATABASE MANAGEMENT SYSTEMS

Outline

The Need for Databases

Data Models

Relational Databases

Database Design

Storage Manager

Query Processing

Transaction Manager

Database Management System (DBMS)

DBMS contains information about a particular enterprise

- Collection of interrelated data
- Set of programs to access the data
- An environment that is both *convenient* and *efficient* to use

Database Applications:

- Banking: transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Sales: customers, products, purchases
- Online retailers: order tracking, customized recommendations
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions

Databases can be very large.

Databases touch all aspects of our lives

University Database Example

Application program examples

- Add new students, instructors, and courses
- Register students for courses, and generate class rosters
- Assign grades to students, compute grade point averages (GPA) and generate transcripts

In the early days, database applications were built directly on top of file systems

Drawbacks of using file systems to store data

Data redundancy and inconsistency

- Multiple file formats, duplication of information in different files

Difficulty in accessing data

- Need to write a new program to carry out each new task

Data isolation

- Multiple files and formats

Integrity problems

- Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
- Hard to add new constraints or change existing ones

Drawbacks of using file systems to store data (Cont.)

Atomicity of updates

- Failures may leave database in an inconsistent state with partial updates carried out
- Example: Transfer of funds from one account to another should either complete or not happen at all

Concurrent access by multiple users

- Concurrent access needed for performance
- Uncontrolled concurrent accesses can lead to inconsistencies
  - Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time

Security problems

- Hard to provide user access to some, but not all, data

Database systems offer solutions to all the above problems

Levels of Abstraction

Physical level: describes how a record (e.g., instructor) is stored.

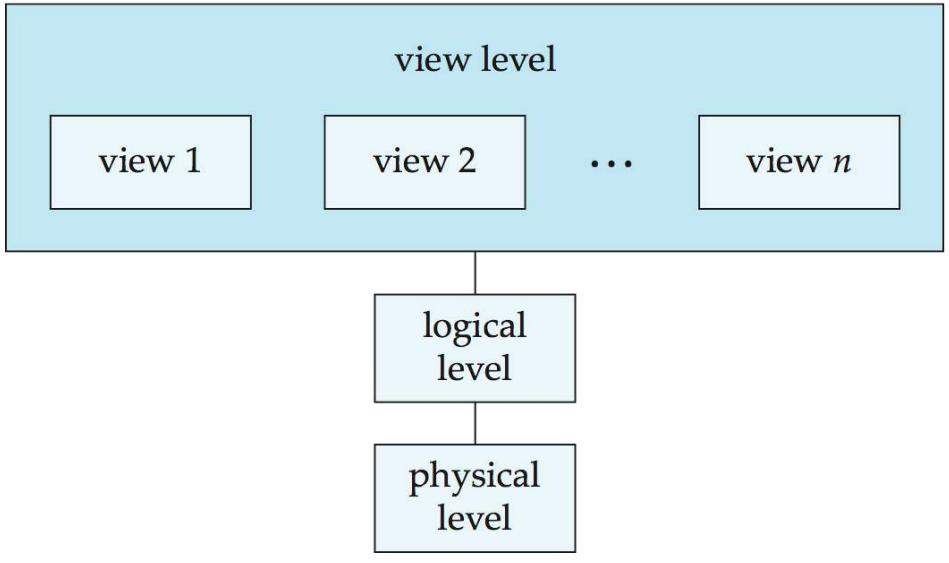
Logical level: describes data stored in database, and the relationships among the data.

```
type instructor = record
                  ID : string;
                  name : string;
                  dept_name : string;
                  salary : integer;
    end;
```

View level: application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

View of Data

An architecture for a database system



Instances and Schemas

Similar to types and variables in programming languages

Logical Schema – the overall logical structure of the database

- Example: The database consists of information about a set of customers and accounts in a bank and the relationship between them
  - Analogous to type information of a variable in a program

Physical schema– the overall physical structure of the database

Instance – the actual content of the database at a particular point in time

- Analogous to the value of a variable

Physical Data Independence – the ability to modify the physical schema without changing the logical schema

- Applications depend on the logical schema
- In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

Data Models

A collection of tools for describing
- Data
- Data relationships
- Data semantics
- Data constraints

Relational model

Entity-Relationship data model (mainly for database design)

Object-based data models (Object-oriented and Object-relational)

Semistructured data model (XML)

Other older models:
- Network model
- Hierarchical model

Relational Model

All the data is stored in various tables.

Example of tabular data in the relational model

Columns

Rows

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

A Sample Relational Database

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

Data Definition Language (DDL)

Specification notation for defining the database schema

Example:

```
create table instructor (
    ID          char(5),
    name        varchar(20),
    dept_name   varchar(20),
    salary      numeric(8,2))
```

DDL compiler generates a set of table templates stored in a ***data dictionary***

Data dictionary contains metadata (i.e., data about data)

- Database schema
- Integrity constraints
  - Primary key (ID uniquely identifies instructors)
- Authorization
  - Who can access what

Data Manipulation Language (DML)

Language for accessing and manipulating the data organized by the appropriate data model

- DML also known as query language

Two classes of languages

- **Pure** – used for proving properties about computational power and for optimization
  - Relational Algebra
  - Tuple relational calculus
  - Domain relational calculus
- **Commercial** – used in commercial systems
  - SQL is the most widely used commercial language

SQL

The most widely used commercial language

SQL is NOT a Turing machine equivalent language

To be able to compute complex functions SQL is usually embedded in some higher-level language

Application programs generally access databases through one of

- Language extensions to allow embedded SQL
- Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database

Database Design

The process of designing the general structure of the database:

Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.

- Business decision – What attributes should we record in the database?
- Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?

Physical Design – Deciding on the physical layout of the database

Database Design (Cont.)

Is there any problem with this relation?

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Design Approaches

Need to come up with a methodology to ensure that each of the relations in the database is “good”

Two ways of doing so:

- Entity Relationship Model
  - Models an enterprise as a collection of *entities* and *relationships*
  - Represented diagrammatically by an *entity-relationship diagram:*
- Normalization Theory
  - Formalize what designs are bad, and test for them

Object-Relational Data Models

Relational model: flat, “atomic” values

Object Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Provide upward compatibility with existing relational languages.

XML: Extensible Markup Language

Defined by the WWW Consortium (W3C)

Originally intended as a document markup language not a database language

The ability to specify new tags, and to create nested tag structures made XML a great way to exchange **data**, not just documents

XML has become the basis for all new generation data interchange formats.

A wide variety of tools is available for parsing, browsing and querying XML documents/data

Database Engine

Storage manager

Query processing

Transaction manager

Storage Management

Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

The storage manager is responsible to the following tasks:

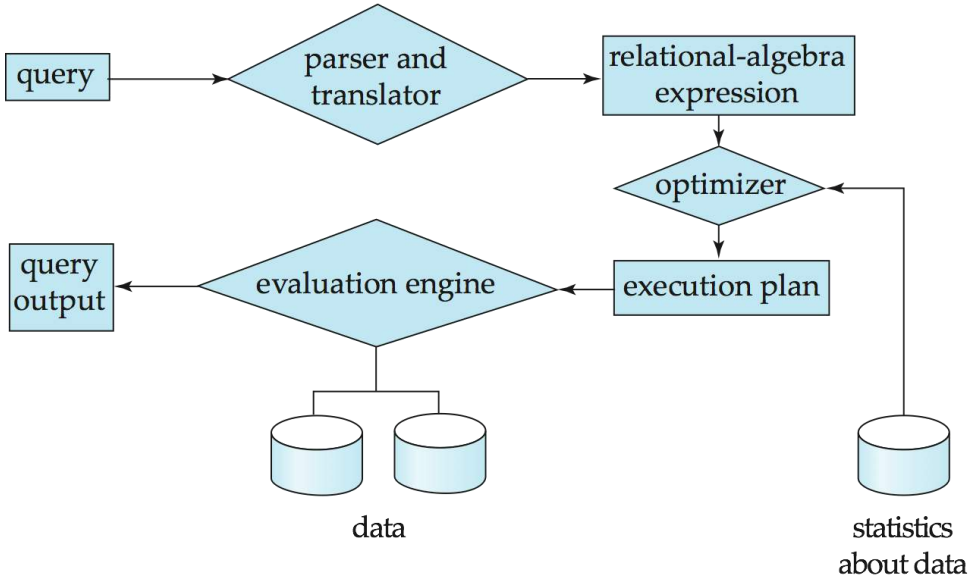
- Interaction with the OS file manager
- Efficient storing, retrieving and updating of data

Issues:

- Storage access
- File organization
- Indexing and hashing

Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Query Processing (Cont.)

Alternative ways of evaluating a given query

- Equivalent expressions
- Different algorithms for each operation

Cost difference between a good and a bad way of evaluating a query can be enormous

Need to estimate the cost of operations

- Depends critically on statistical information about relations which the database must maintain
- Need to estimate statistics for intermediate results to compute cost of complex expressions

Transaction Management

What if the system fails?

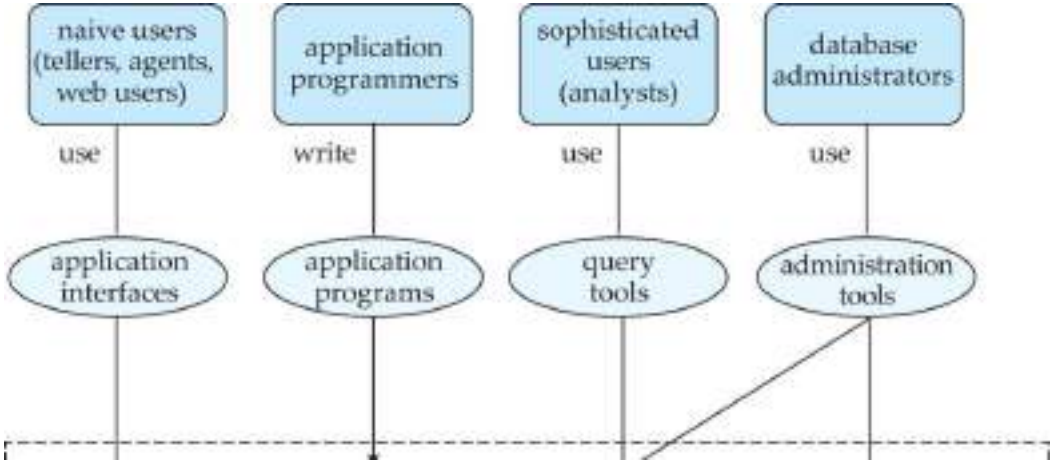
What if more than one user is concurrently updating the same data?

A **transaction** is a collection of operations that performs a single logical function in a database application

Transaction-management component ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

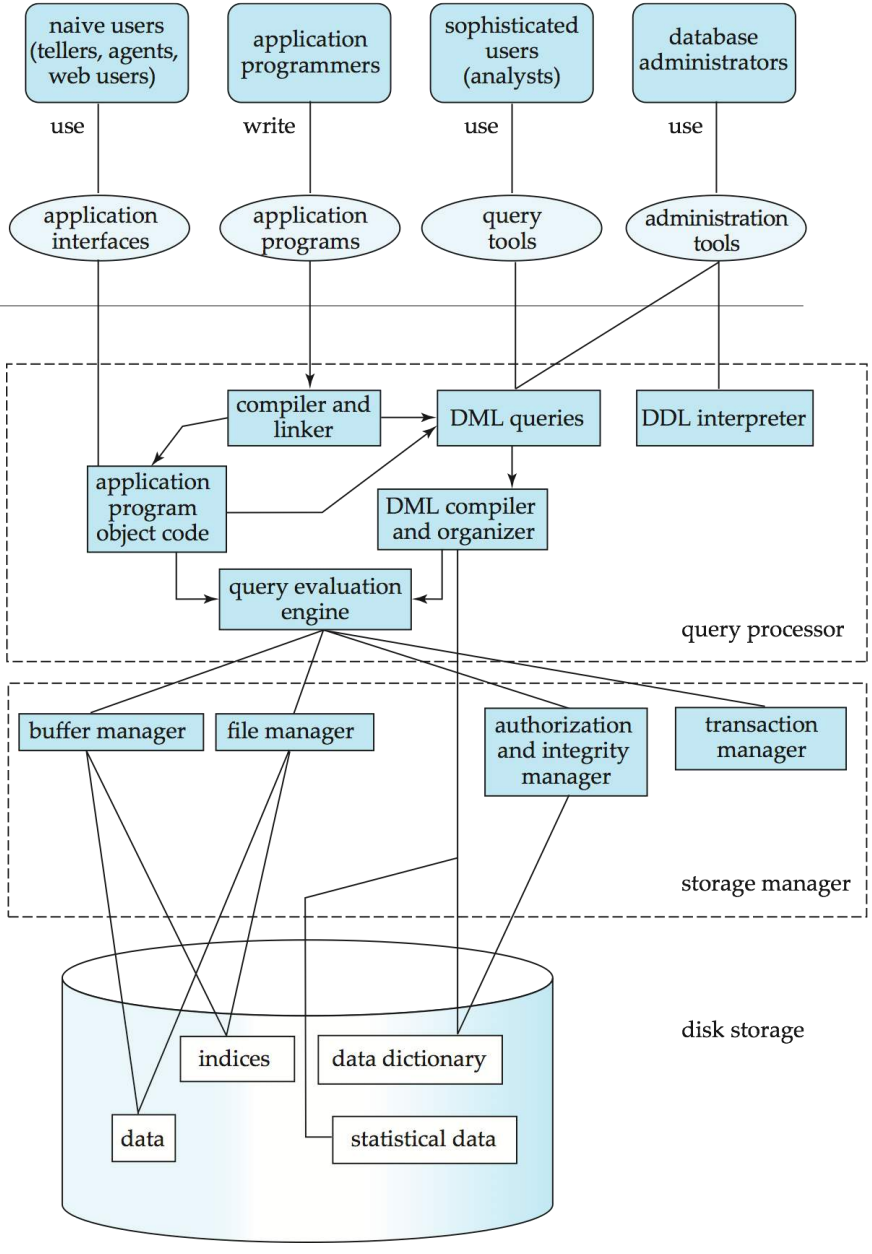
Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Database Users and Administrators



Database

Database System Internals



Database Architecture

The architecture of a database systems is greatly influenced by the underlying computer system on which the database is running:

Centralized

Client-server

Parallel (multi-processor)

Distributed

History of Database Systems

1950s and early 1960s:

- Data processing using magnetic tapes for storage
  - Tapes provided only sequential access
- Punched cards for input

Late 1960s and 1970s:

- Hard disks allowed direct access to data
- Network and hierarchical data models in widespread use
- Ted Codd defines the relational data model
  - Would win the ACM Turing Award for this work
  - IBM Research begins System R prototype
  - UC Berkeley begins Ingres prototype
- High-performance (for the era) transaction processing

History (cont.)

1980s:

- Research relational prototypes evolve into commercial systems
  - SQL becomes industrial standard
- Parallel and distributed database systems
- Object-oriented database systems

1990s:

- Large decision support and data-mining applications
- Large multi-terabyte data warehouses
- Emergence of Web commerce

Early 2000s:

- XML and XQuery standards
- Automated database administration

History (cont.)

Later 2000s:

- Giant data storage systems
  - Google BigTable, Yahoo PNuts, Amazon, ..
- NoSQL systems
  - MongoDB, Casandra, CouchDB, Neo4j, HBASE, …

The E/R Model

BBM471 Database Management Systems

Today's Lecture

1. E/R Basics: Entities & Relations
2. E/R Design considerations
3. Advanced E/R Concepts

1. E/R Basics: Entities & Relations

What you will learn about in this section

1. High-level motivation for the E/R model
2. Entities
3. Relations

Database Design

- **Database design: Why do we need it?**
  - Agree on structure of the database before deciding on a particular implementation

- **Consider issues such as:**
  - What entities to model
  - How entities are related
  - What constraints exist in the domain
  - How to achieve <u>good</u> designs

- **Several formalisms exist**
  - We discuss flavors of E/R diagrams

Database Design Process

1. Requirements Analysis → 2. Conceptual Design → 3. Logical, Physical, Security, etc.

1. Requirements analysis

- What is going to be stored?
- How is it going to be used?
- What are we going to do with the data?
- Who should access the data?

Technical and non-technical people are involved

Database Design Process

1. Requirements Analysis → **2. Conceptual Design** → 3. Logical, Physical, Security, etc.

2. Conceptual Design

- A high-level description of the database
- Sufficiently precise that technical people can understand it
- But, not so precise that non-technical people can't participate

This is where E/R fits in.

Database Design Process

1. Requirements Analysis → 2. Conceptual Design → 3. Logical, Physical, Security, etc.

3. More:

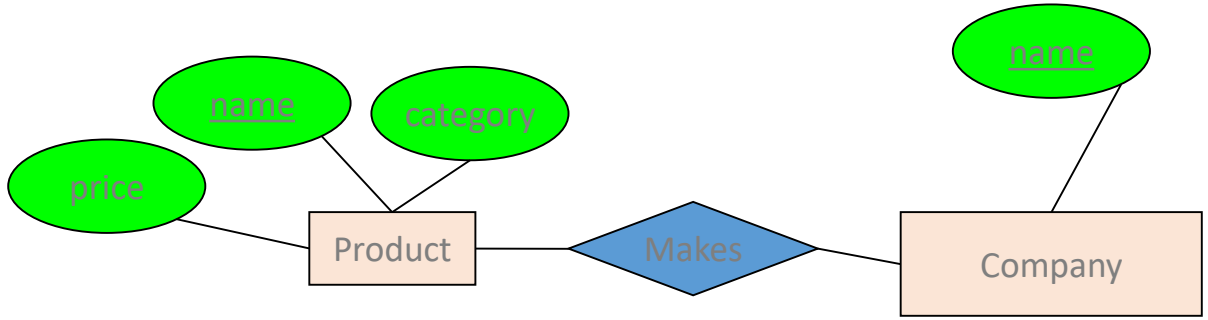
- Logical Database Design
- Physical Database Design
- Security Design

Database Design Process

1. Requirements Analysis → 2. Conceptual Design → 3. Logical, Physical, Security, etc.

E/R Model & Diagrams used

This process is iterated **many** times



E/R is a *visual syntax* for DB design which is ***precise enough*** for technical points, but ***abstracted enough*** for non-technical people

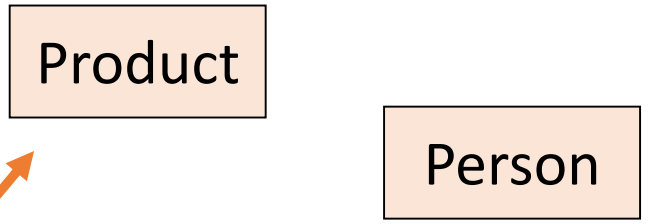
Interlude: Impact of the ER model

- The E/R model is one of the most cited articles in Computer Science
  - *"The Entity-Relationship model – toward a unified view of data"* Peter Chen, 1976

- Used by companies big and small
  - You'll know it soon enough

Entities and Entity Sets

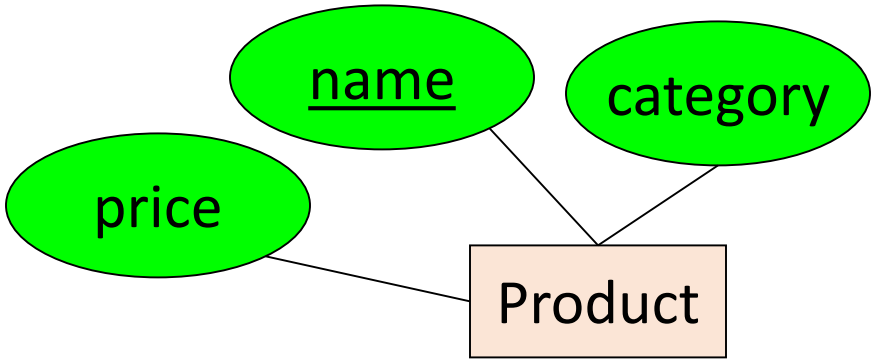
- **Entities** & **entity sets** are the primitive unit of the E/R model
  - Entities are the individual objects, which are members of entity sets
    - Ex: A specific person or product
  - Entity sets are the *classes* or *types* of objects in our model
    - Ex: Person, Product
    - *These are what is shown in E/R diagrams - as rectangles*
    - *Entity sets represent the sets of all possible entities*



These represent entity sets

Entities and Entity Sets

- An entity set has **attributes**
  - Represented by ovals attached to an entity set

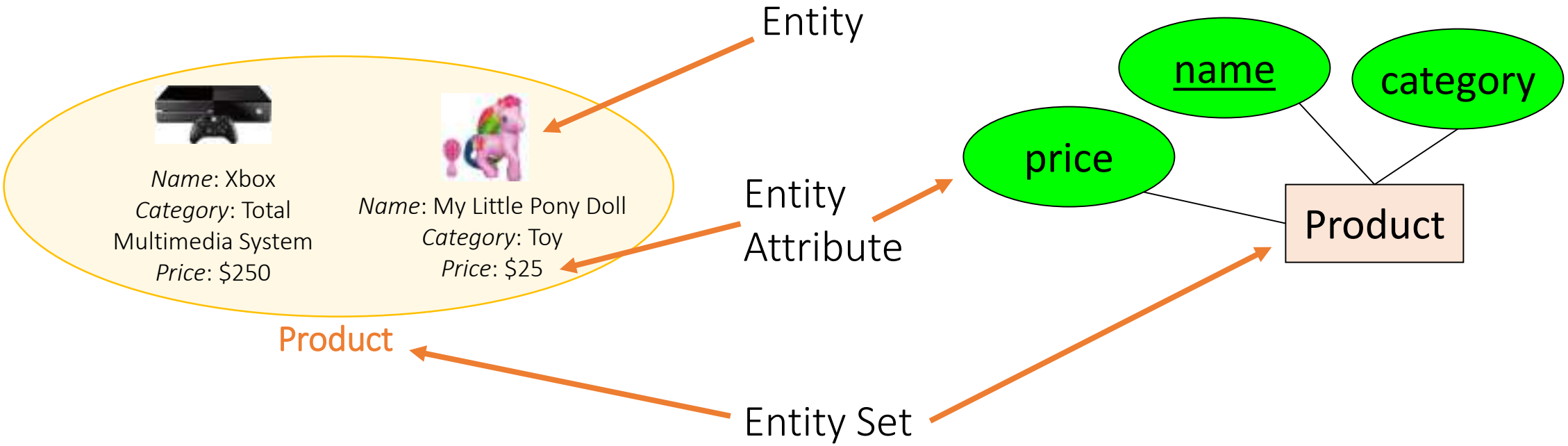


Shapes **are** important.
Colors **are not**.

Entities vs. Entity Sets

Example:

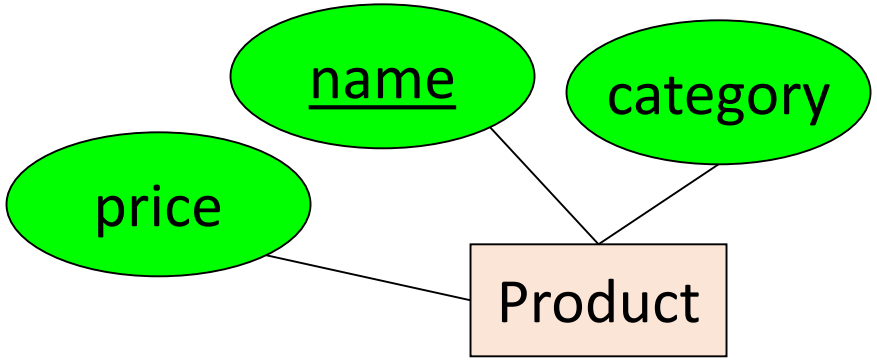
Entities are **not** explicitly represented in E/R diagrams!



Keys

- A *key* is a **minimal** set of attributes that uniquely identifies an entity.

Denote elements of the primary key by underlining.



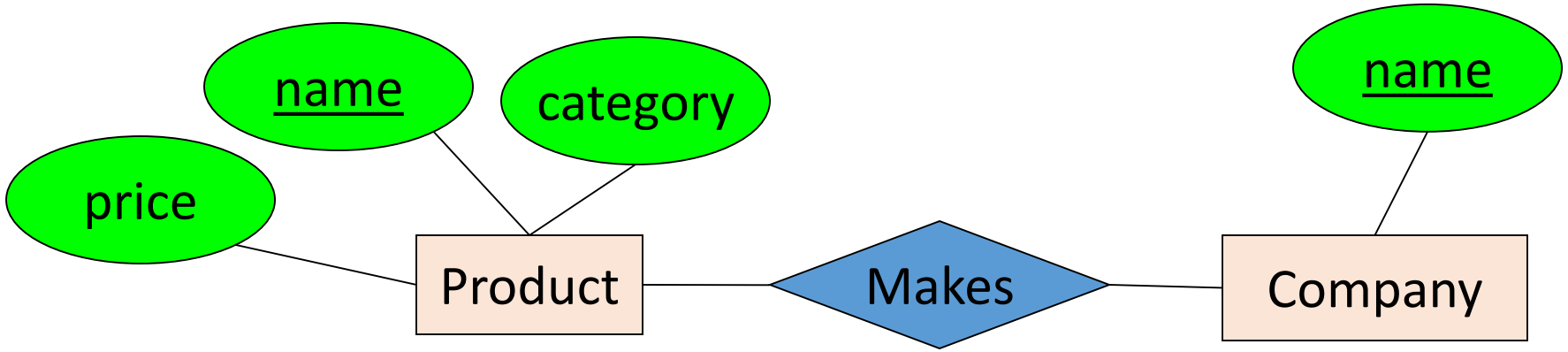
Here, {price, category} is **not** a key.

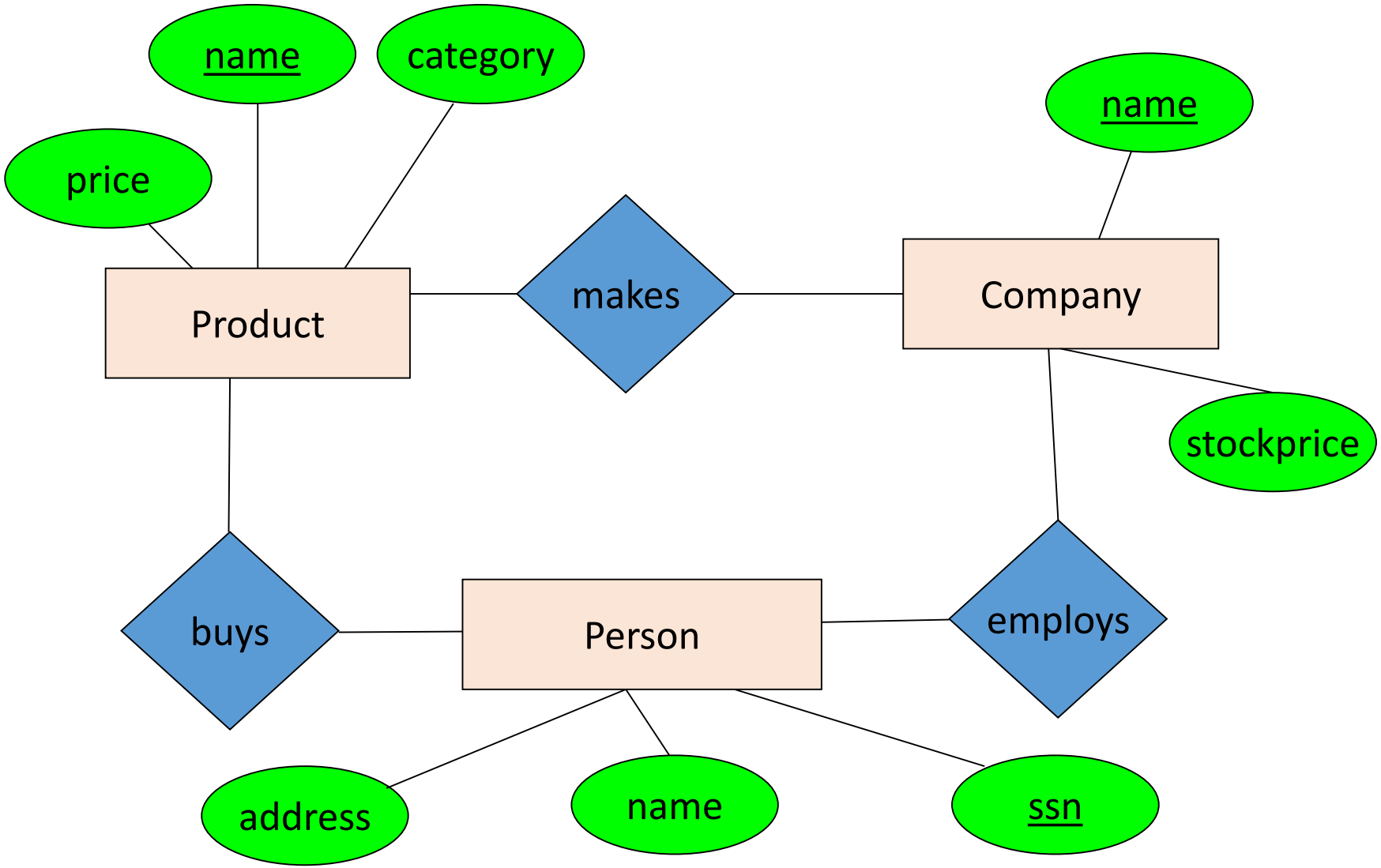
If it were, what would it mean?

The E/R model forces us to designate a single **primary** key, though there may be multiple candidate keys

The R in E/R: **Relationships**

- A **relationship** is between two entities

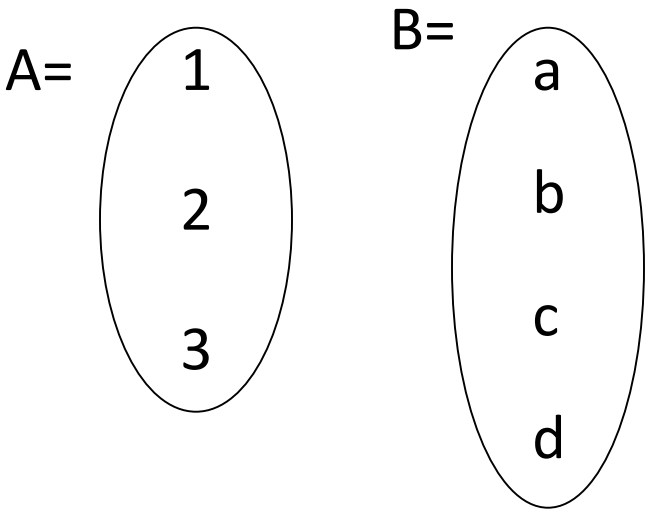




Company makes one product, employs one person.
Person buys one product.

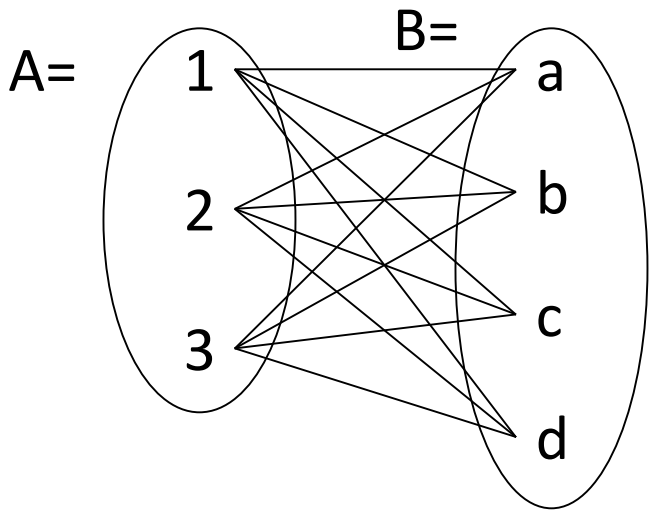
What is a Relationship?

- ***A mathematical definition:***
  - Let A, B be sets
    - *A={1,2,3}, B={a,b,c,d}*



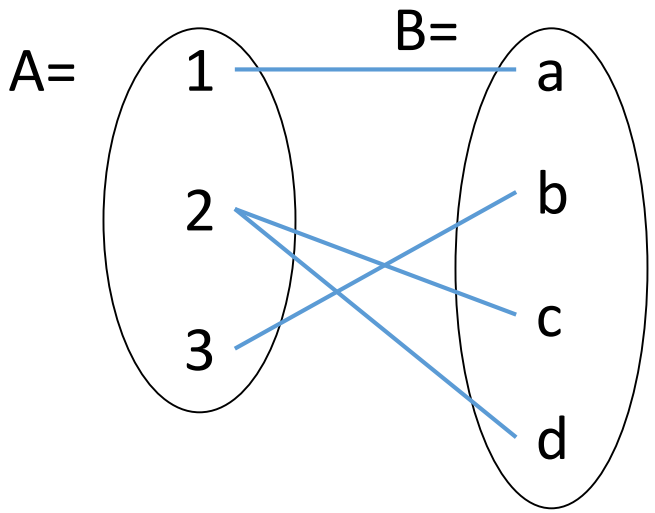
What is a Relationship?

- ***A mathematical definition:***
  - Let A, B be sets
    - *A={1,2,3}, B={a,b,c,d}*
  - A x B (the ***cross-product***) is the set of all pairs (a,b)
    - *A $\times$ B = {(1,a), (1,b), (1,c), (1,d), (2,a), (2,b), (2,c), (2,d), (3,a), (3,b), (3,c), (3,d)}*



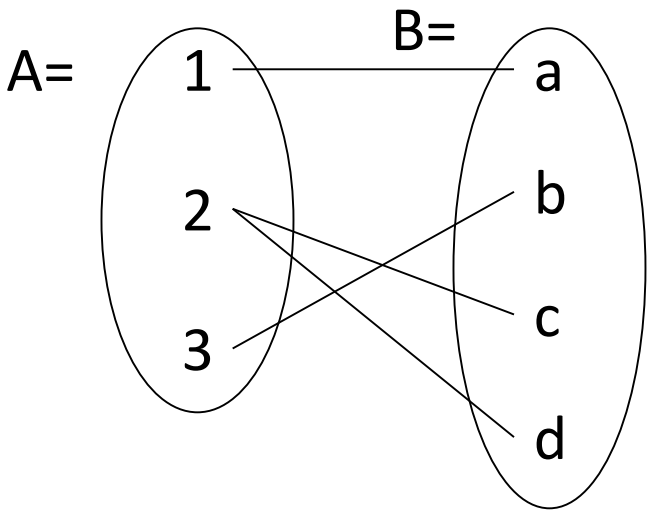
What is a Relationship?

- ***A mathematical definition:***
  - Let A, B be sets
    - *A={1,2,3}, B={a,b,c,d},*
  - A x B (the ***cross-product***) is the set of all pairs (a,b)
    - *A × B = {(1,a), (1,b), (1,c), (1,d), (2,a), (2,b), (2,c), (2,d), (3,a), (3,b), (3,c), (3,d)}*
  - **We define a <u>relationship</u> to be a subset of A x B**
    - *R = {(1,a), (2,c), (2,d), (3,b)}*



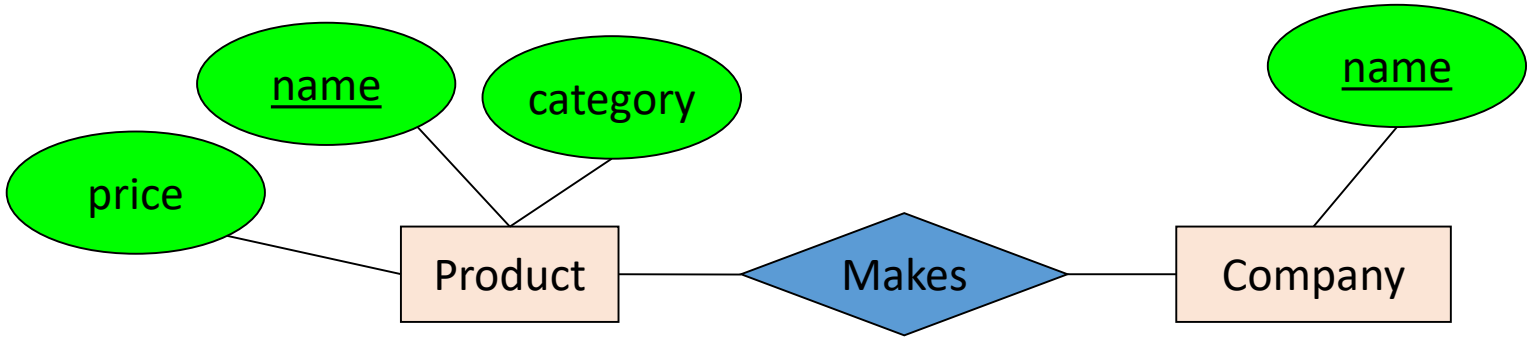
What is a Relationship?

- ***A mathematical definition:***
  - Let A, B be sets
  - A x B (the ***cross-product***) is the set of all pairs
  - A <u>relationship</u> is a subset of A x B



- **Makes** is relationship- it is a ***subset*** of **Product × Company**:

What is a Relationship?



A <u>**relationship**</u> between **entity sets P and C** is a ***subset of all possible pairs of entities in P and C***, with tuples uniquely identified by ***P and C's keys***

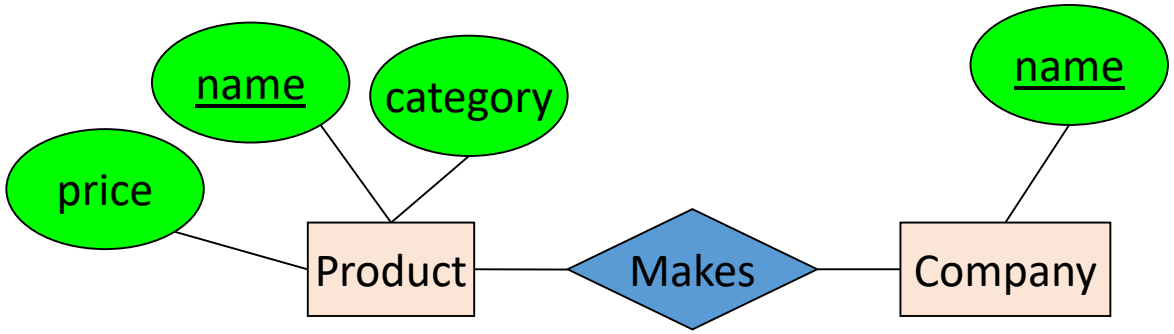
What is a Relationship?

Company

name
GizmoWorks
GadgetCorp

Product

name	category	price
Gizmo	Electronics	$9.99
GizmoLite	Electronics	$7.50
Gadget	Toys	$5.50



A <u>relationship</u> between **entity sets P and C** is a ***subset of all possible pairs of entities in P and C***, with tuples uniquely identified by ***P and C's keys***

What is a Relationship?

Company

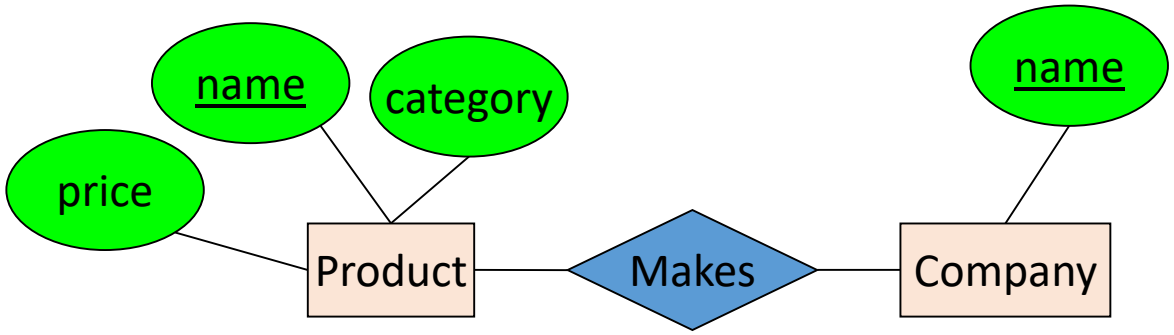
name
GizmoWorks
GadgetCorp

Product

name	category	price
Gizmo	Electronics	$9.99
GizmoLite	Electronics	$7.50
Gadget	Toys	$5.50

Company C × Product P

C.name	P.name	P.category	P.price
GizmoWorks	Gizmo	Electronics	$9.99
GizmoWorks	GizmoLite	Electronics	$7.50
GizmoWorks	Gadget	Toys	$5.50
GadgetCorp	Gizmo	Electronics	$9.99
GadgetCorp	GizmoLite	Electronics	$7.50
GadgetCorp	Gadget	Toys	$5.50



A <u>relationship</u> between **entity sets P and C** is a ***subset of all possible pairs of entities in P and C***, with tuples uniquely identified by ***P and C's keys***

What is a Relationship?

Company

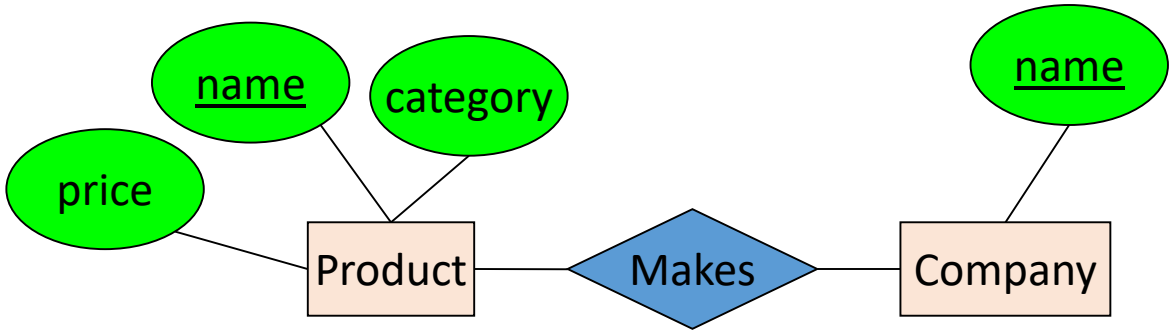
name
GizmoWorks
GadgetCorp

Product

name	category	price
Gizmo	Electronics	$9.99
GizmoLite	Electronics	$7.50
Gadget	Toys	$5.50

Company C × Product P

C.name	P.name	P.category	P.price
GizmoWorks	Gizmo	Electronics	$9.99
GizmoWorks	GizmoLite	Electronics	$7.50
GizmoWorks	Gadget	Toys	$5.50
GadgetCorp	Gizmo	Electronics	$9.99
GadgetCorp	GizmoLite	Electronics	$7.50
GadgetCorp	Gadget	Toys	$5.50



Makes

C.name	P.name
GizmoWorks	Gizmo
GizmoWorks	GizmoLite
GadgetCorp	Gadget

A <u>relationship</u> between **entity sets P and C** is a ***subset of all possible pairs of entities in P and C***, with tuples uniquely identified by ***P and C's keys***

What is a Relationship?

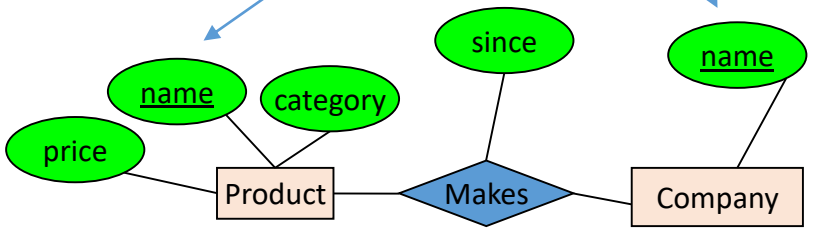
- There can only be **one relationship for every unique combination of entities**

This follows from our mathematical definition of a relationship- it's a SET!

- This also means that **the relationship is uniquely determined by the keys of its entities**

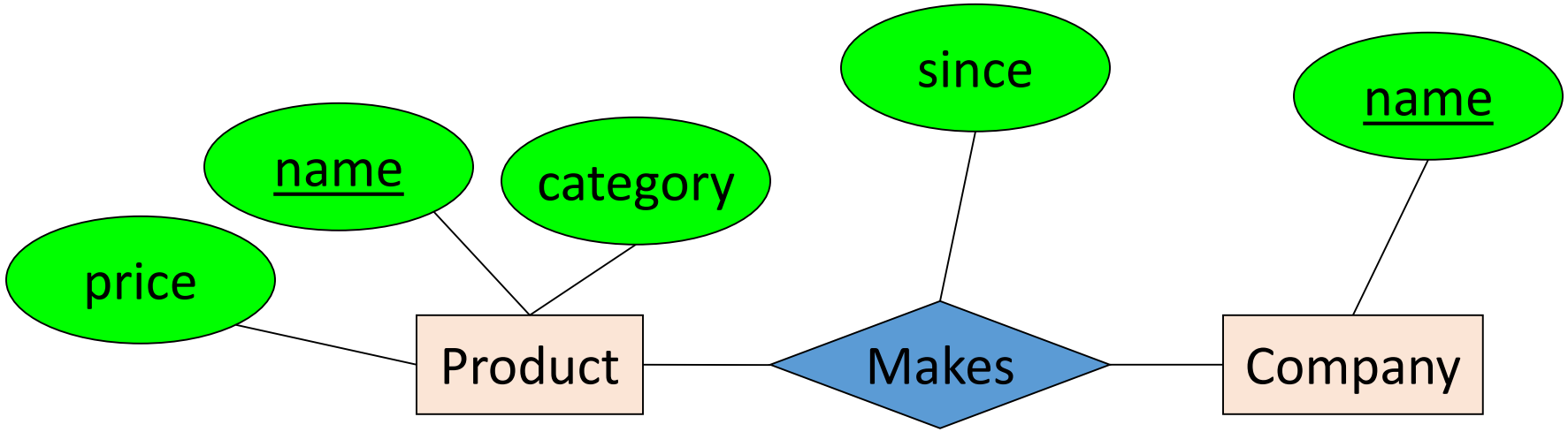
- *Example: the "key" for Makes (to right) is {Product.name, Company.name}*

$Key_{Makes} = Key_{Product} \cup Key_{Company}$



Relationships and Attributes

- Relationships may have attributes as well.



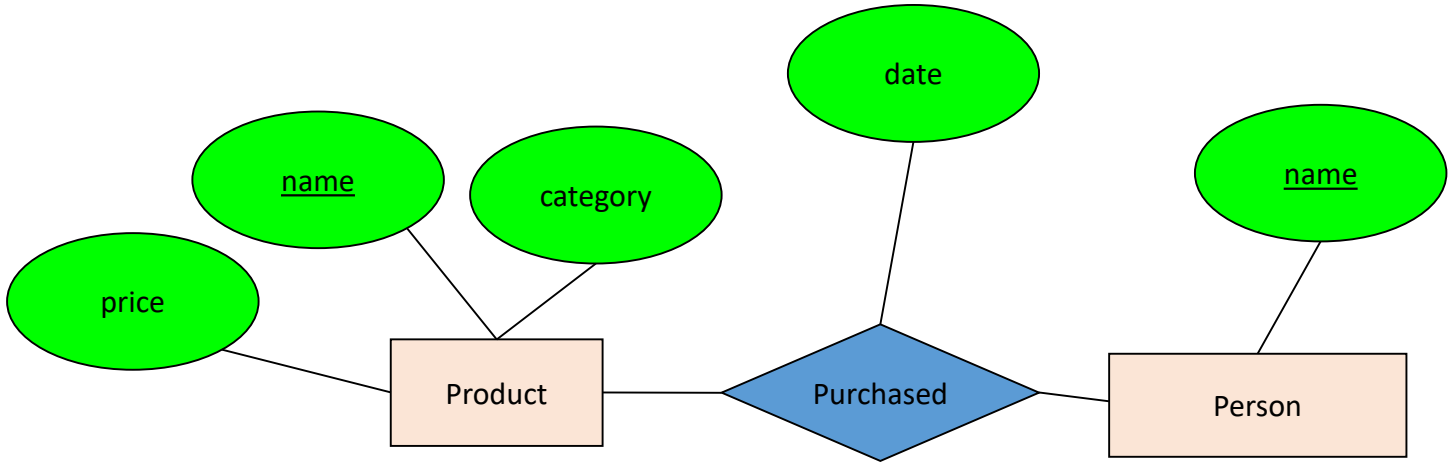
For example: "since" records when company started making a product

Note: *"since"* is implicitly unique per pair here! Why?

Note #2: Why not "how long"?

Decision: Relationship vs. Entity?

- **Q:** What does this say?

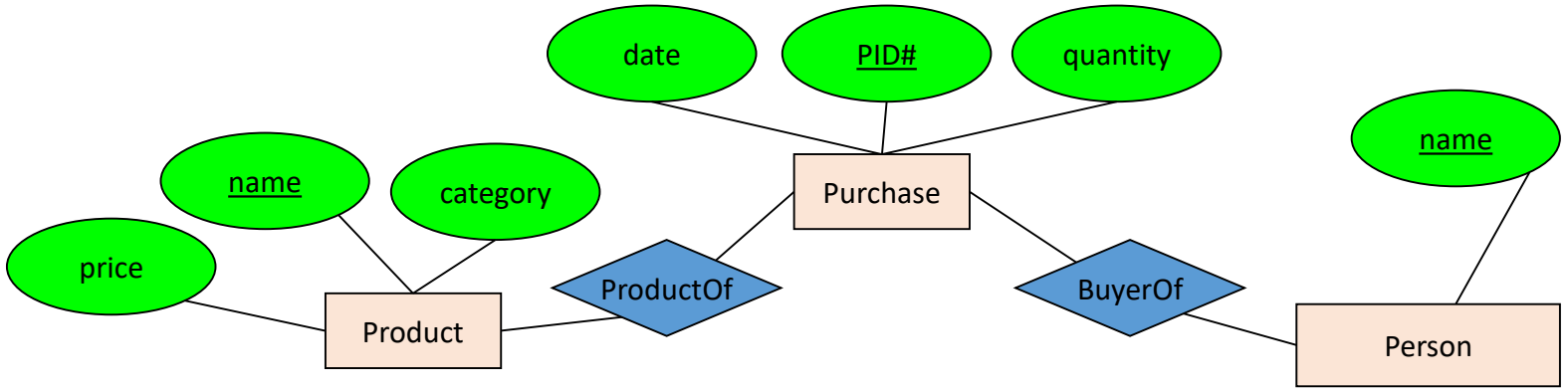


- **A:** A person can only buy a specific product once (on one date)

Modeling something as a relationship makes it unique; what if not appropriate?

Decision: Relationship vs. Entity?

- What about this way?

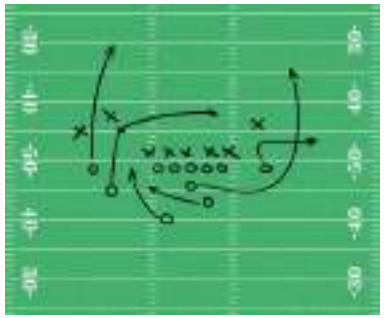


- *Now we can have multiple purchases per product, person pair!*

We can always use **a new entity** instead of a relationship. For example, to permit multiple instances of each entity combination!

Draw an E/R diagram for football

Use the following simplified model of a football season
(concepts to include are underlined):



Teams play each other in **Games**. Each pair of teams can play each other multiple times

Players belong to Teams (assume no trades / changes).

A Game is made up of **Plays** that result in a yardage gain/loss, and potentially a touchdown

A Play will contain either a **Pass** from one player to another, or a **Run** by one player

*https://twitter.com/McBPJ/status/638728908628586496/photo/1

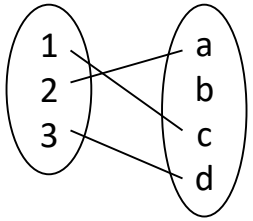
2. E/R Design Considerations

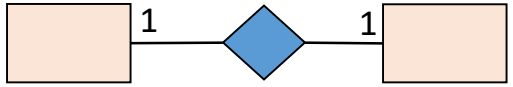
What you will learn about in this section

1. Relationships cont'd: multiplicity, multi-way
2. Design considerations
3. Conversion to SQL

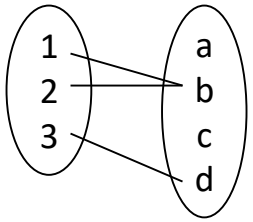
Multiplicity of E/R Relationships

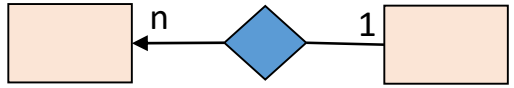
One-to-one:



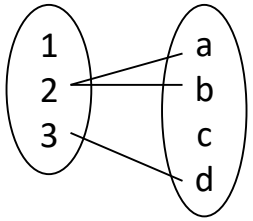


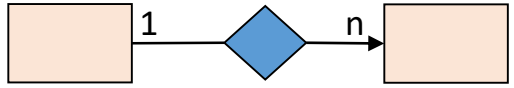
Many-to-one:



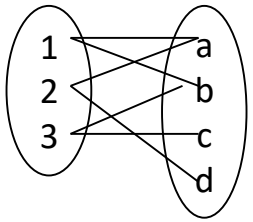


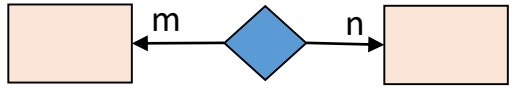
One-to-many:





Many-to-many:

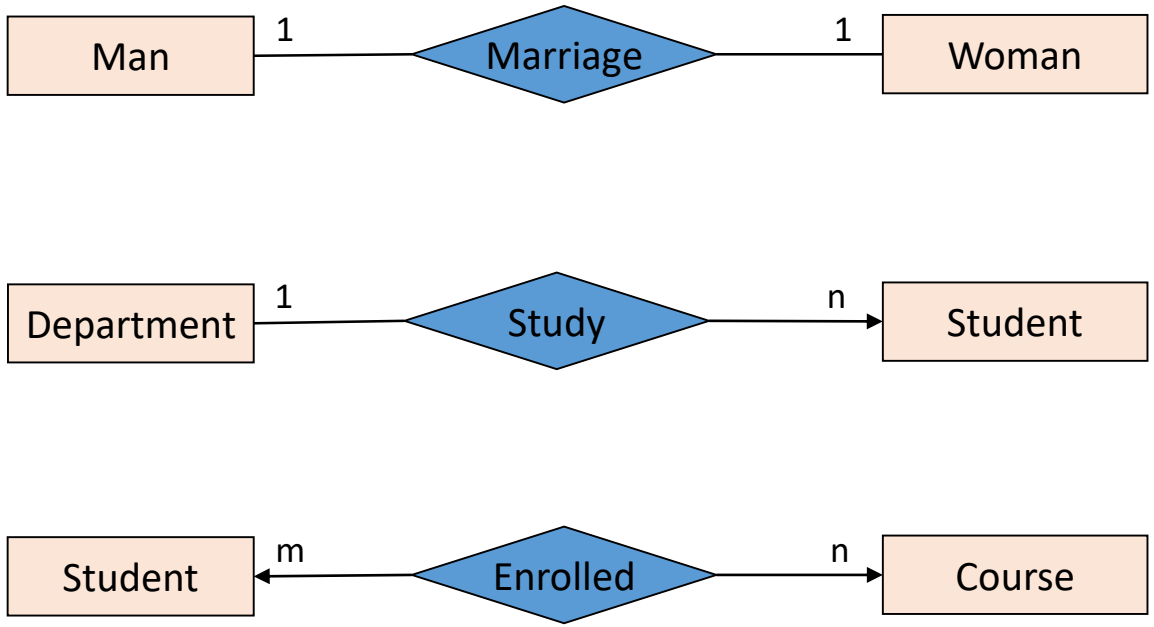


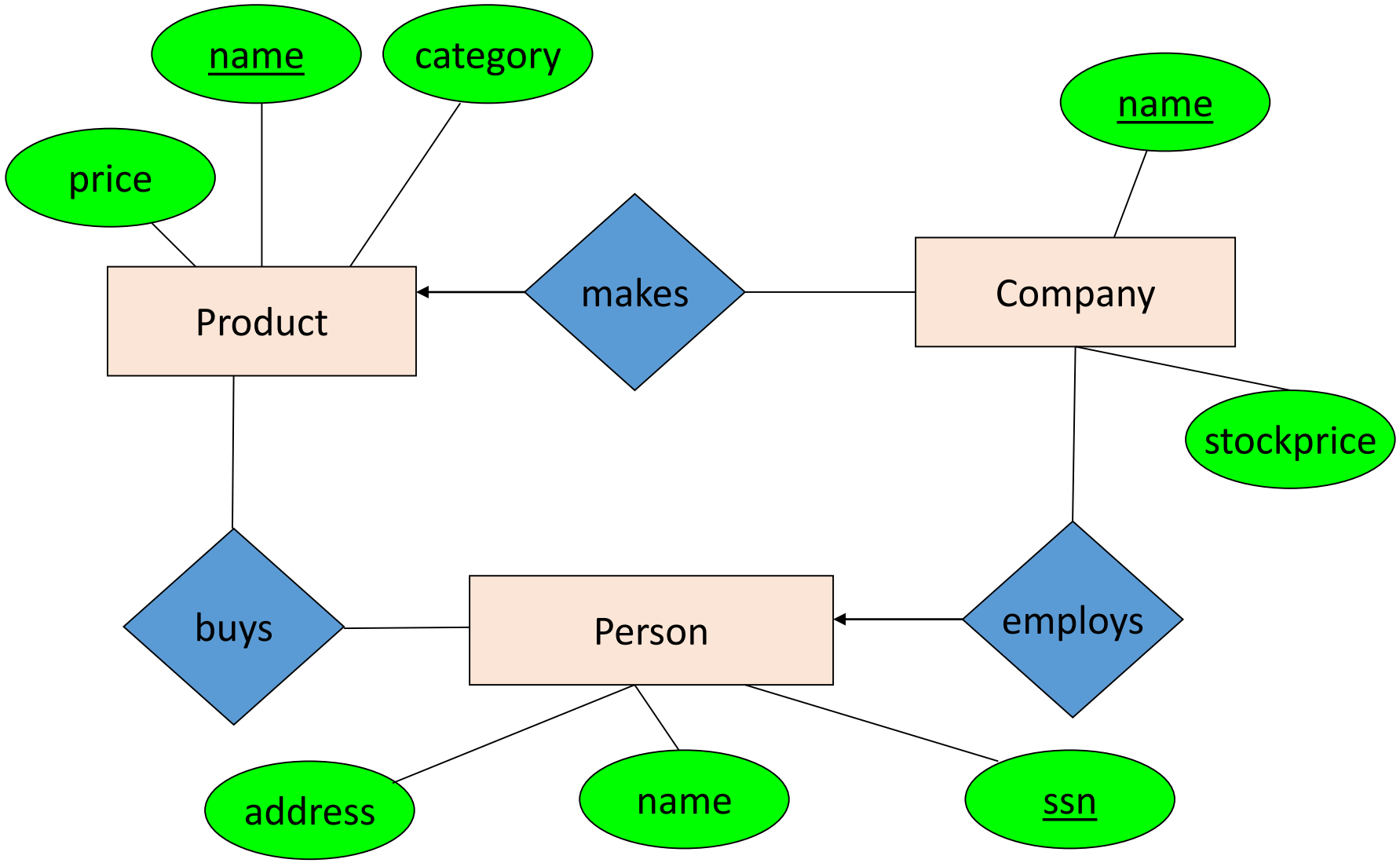


Indicated using arrows

X -> Y means **<u>there exists a function mapping from X to Y</u>** (*recall the definition of a function*)

Multiplicity of E/R Relationships

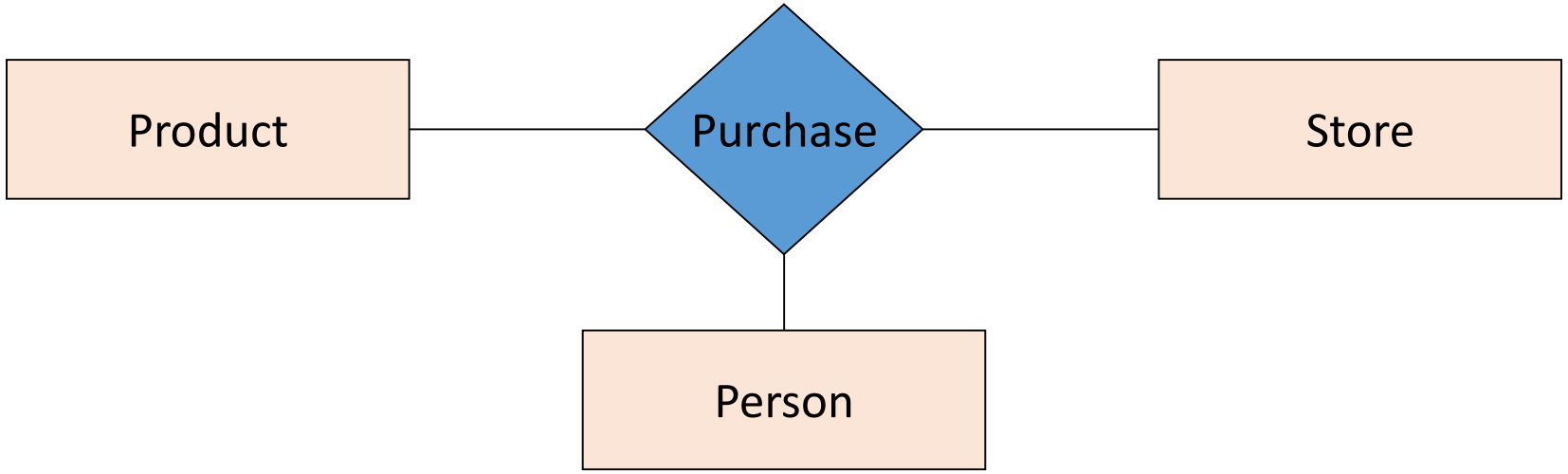




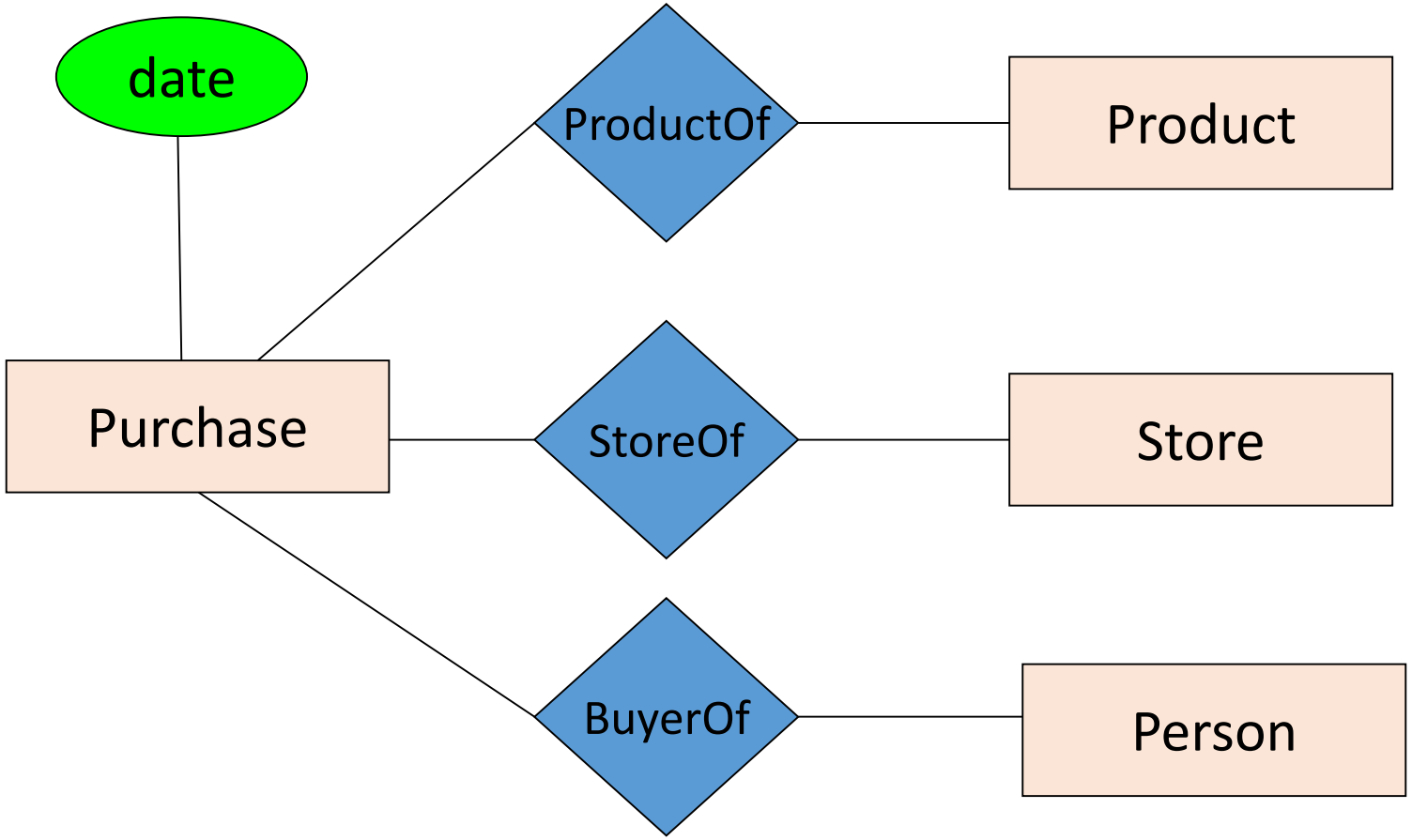
Company can make many product, can employ many person.
Person buys still one product.

Multi-way Relationships

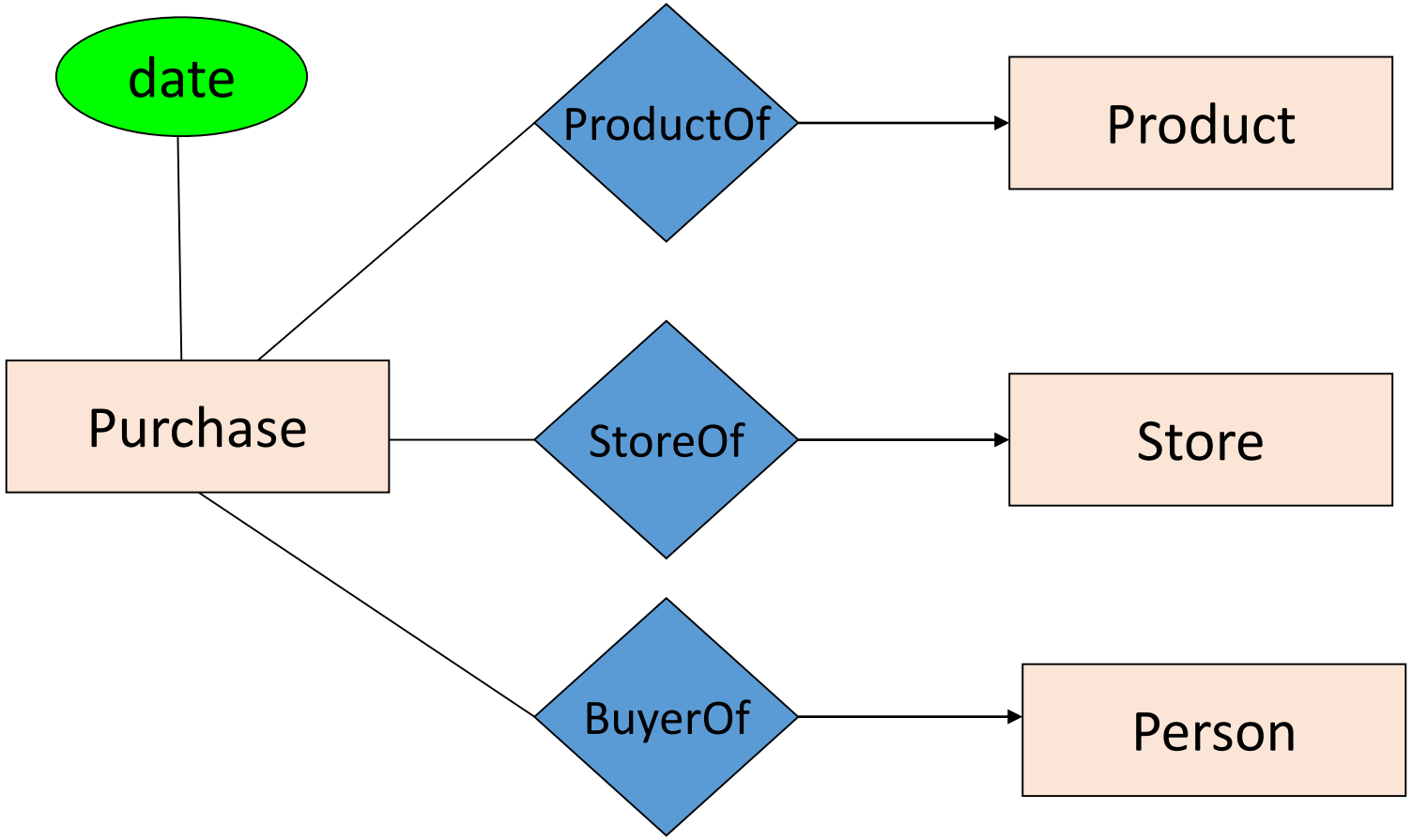
How do we model a purchase relationship between buyers, products and stores?



Converting Multi-way Relationships to Binary

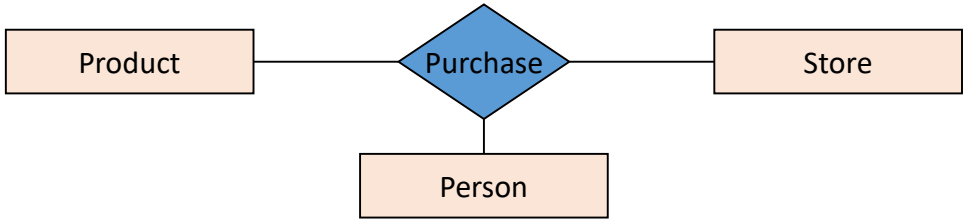


Converting Multi-way Relationships to New Entity + Binary Relationships

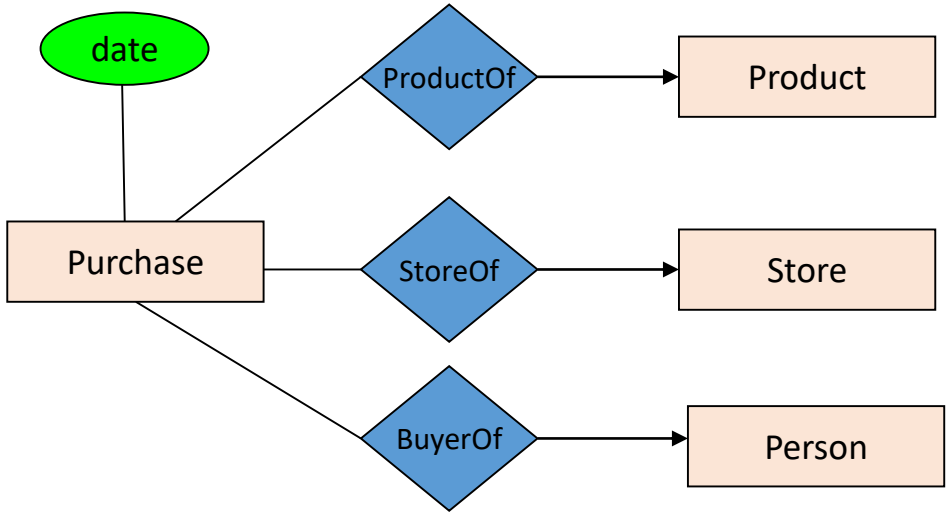


Decision: Multi-way or New Entity + Binary?

(A) Multi-way Relationship



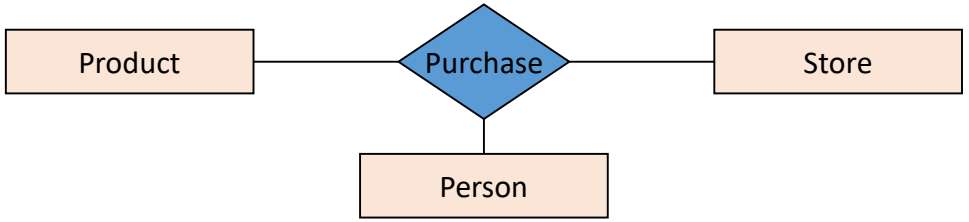
(B) Entity + Binary



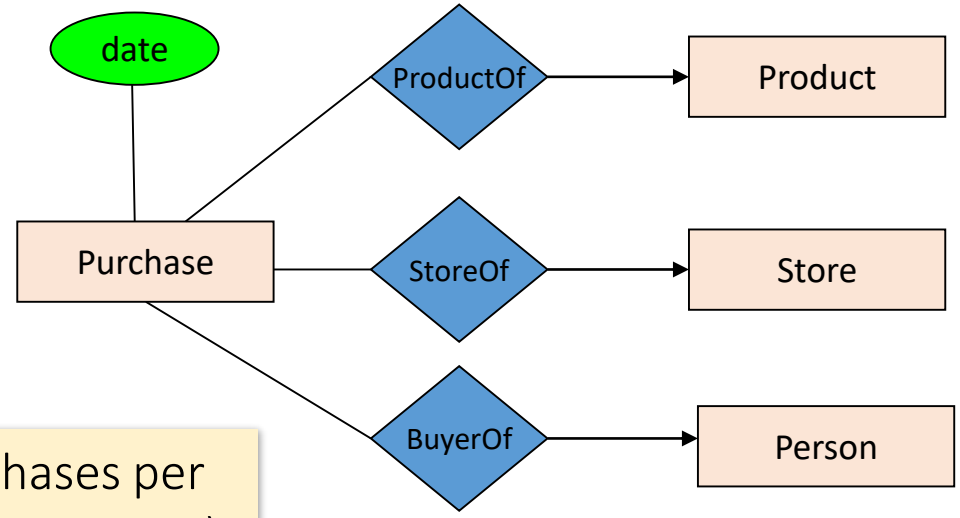
Should we use a single **multi-way relationship** or a ***new entity with binary relations?***

Decision: Multi-way or New Entity + Binary?

(A) Multi-way Relationship



(B) Entity + Binary

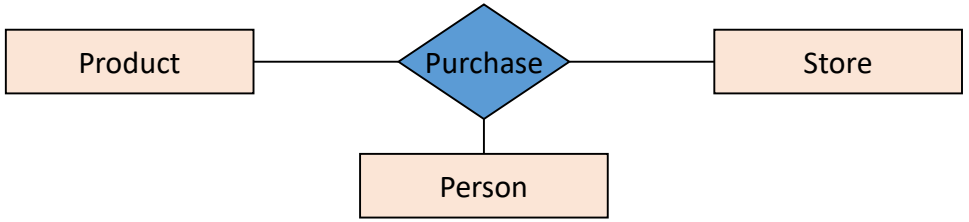


Multiple purchases per (product, store, person) combo possible here!

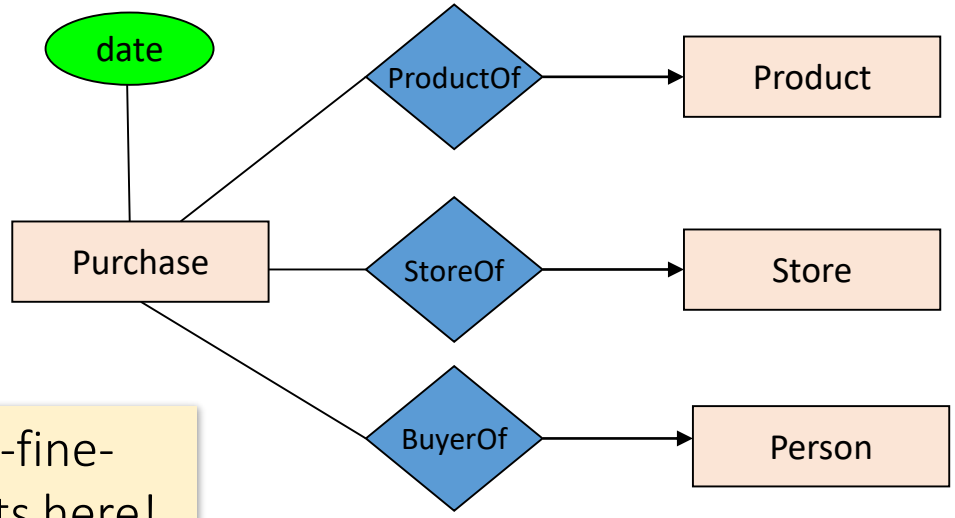
(B) is useful if we want to have multiple instances of the “relationship” per entity combination

Decision: Multi-way or New Entity + Binary?

(A) Multi-way Relationship



(B) Entity + Binary



We can add more-fine-grained constraints here!

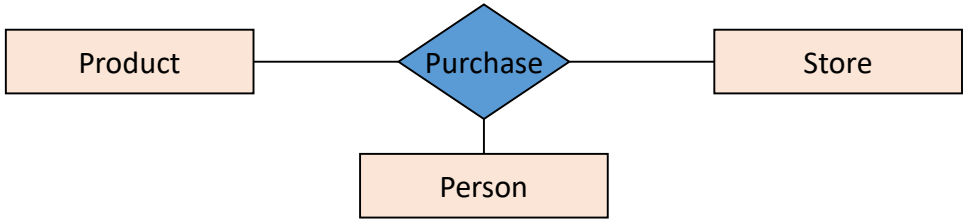
(B) is also useful when we want to add details (constraints or attributes) to the relationship

"A person who shops in only one store"

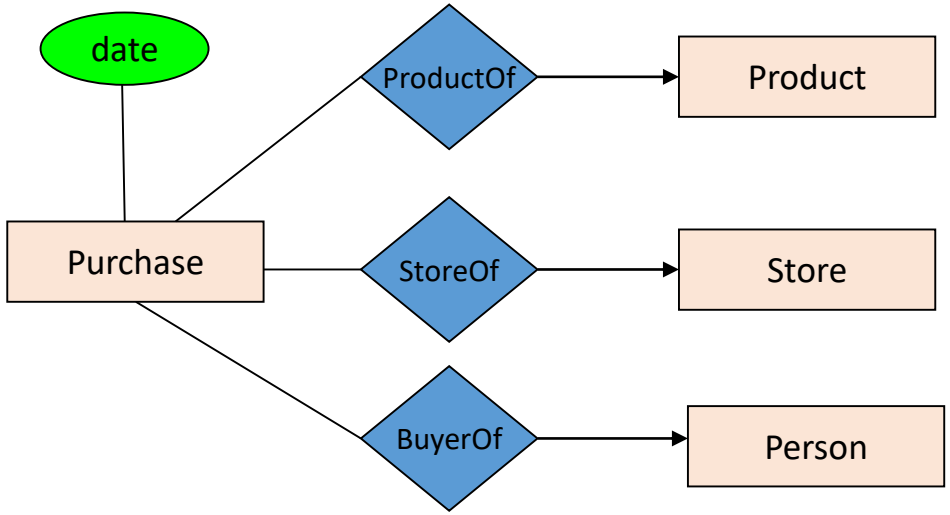
"How long a person has been shopping at a store"

Decision: Multi-way or New Entity + Binary?

(A) Multi-way Relationship



(B) Entity + Binary

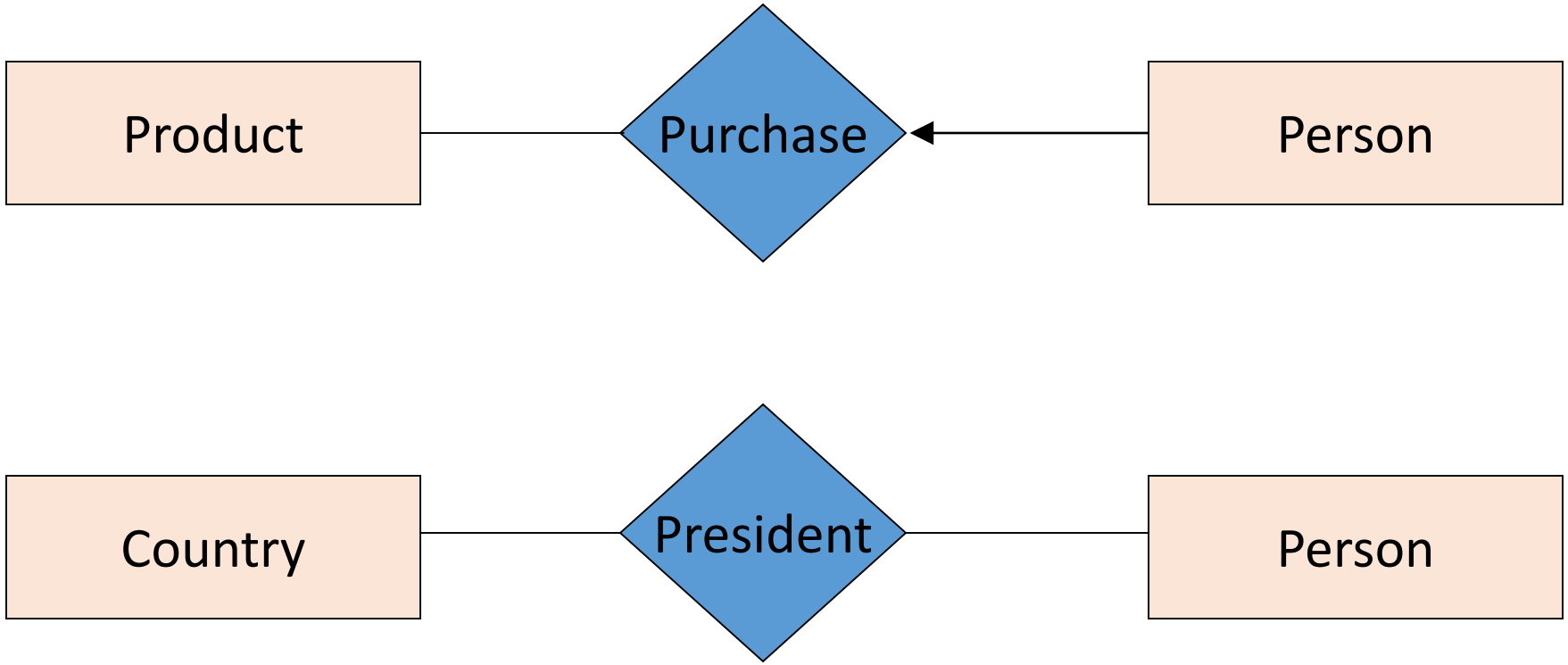


(A) is useful when a relationship really is between multiple entities

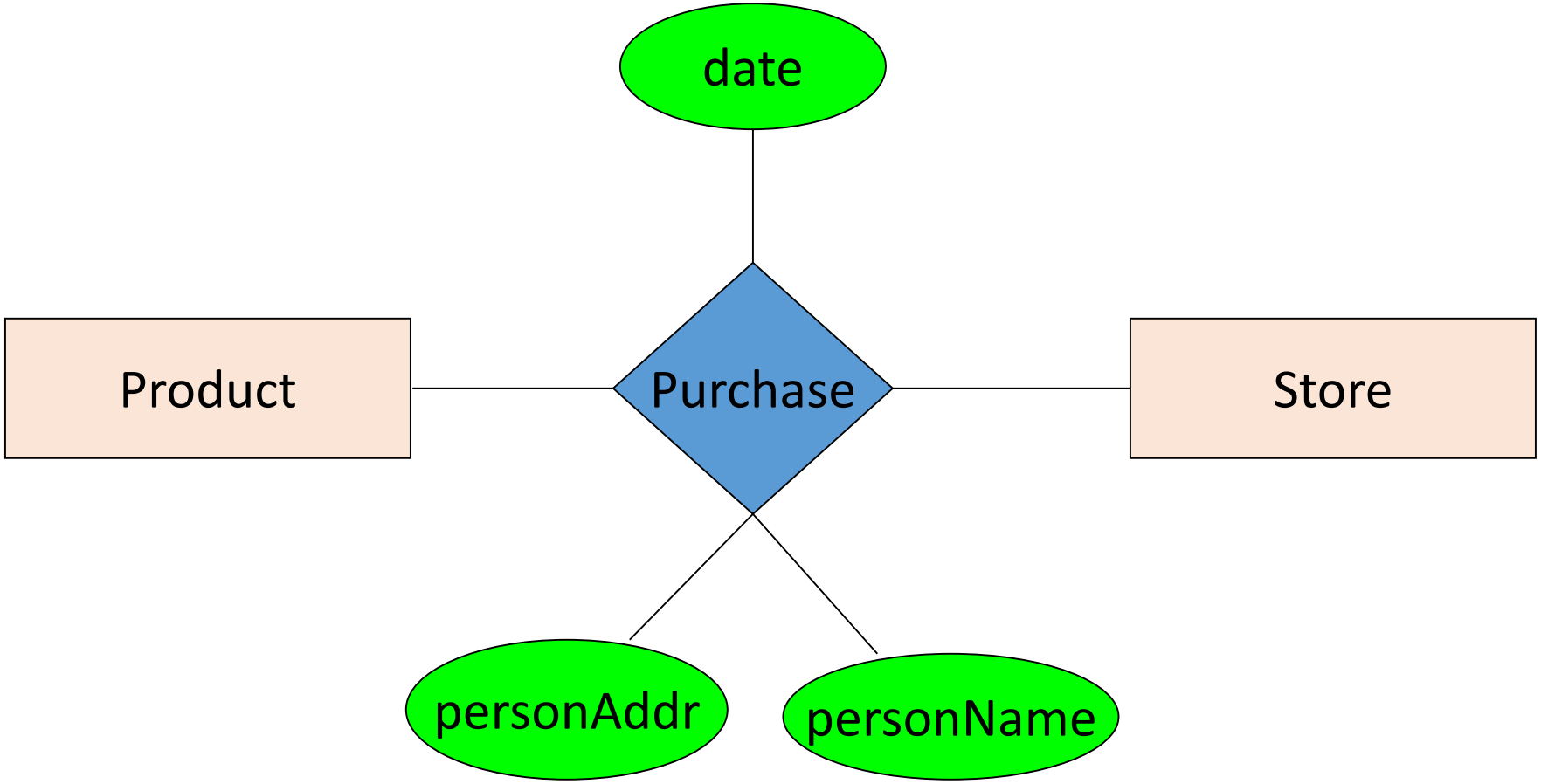
- *Ex: A three-party legal contract*

3. Design Principles

What's wrong with these examples?

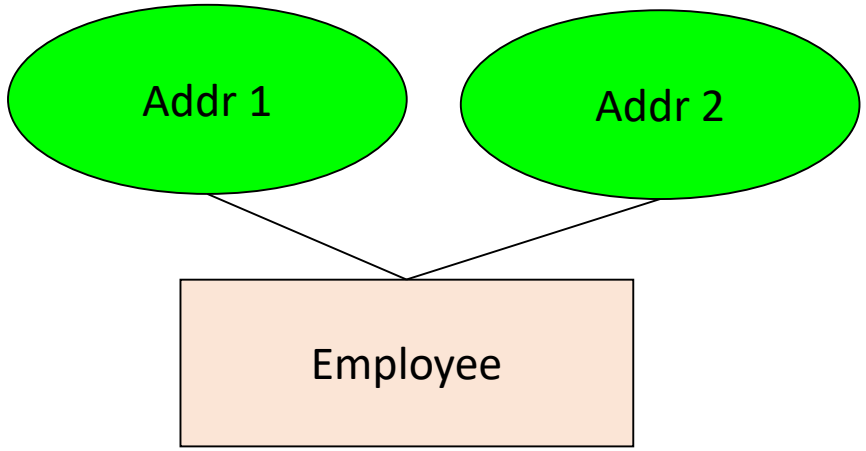


Design Principles: What's Wrong?

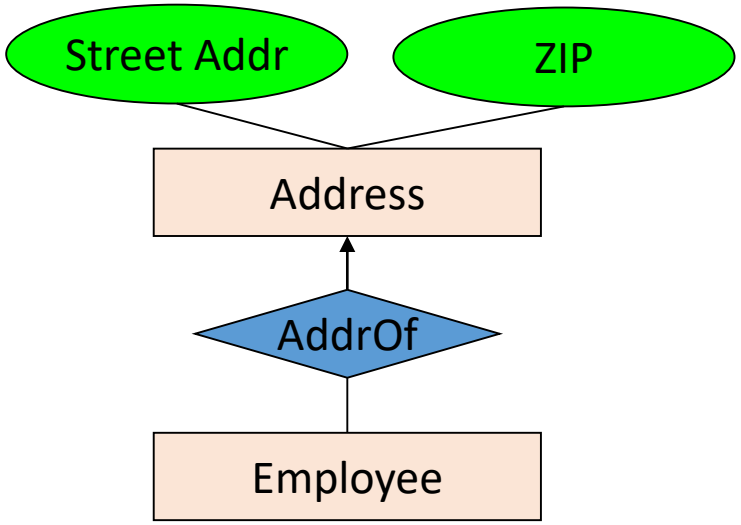


Examples: Entity vs. Attribute

Should address (A) be an attribute?

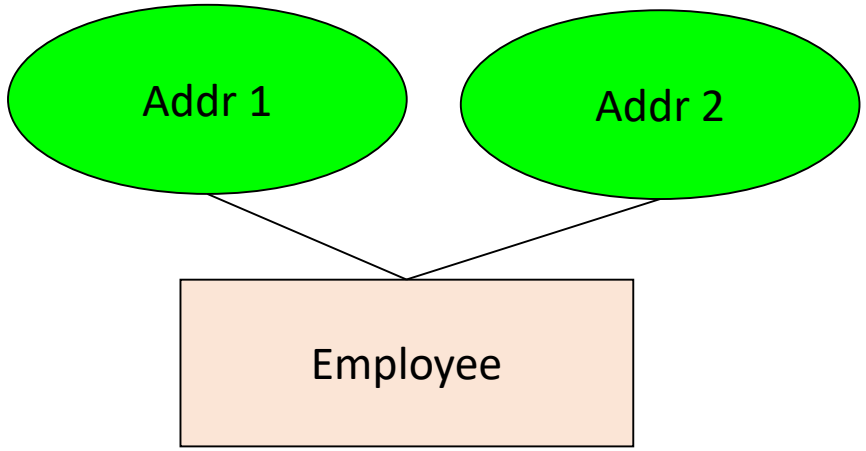


Or (B) be an entity?



Examples: Entity vs. Attribute

Should address (A) be an attribute?

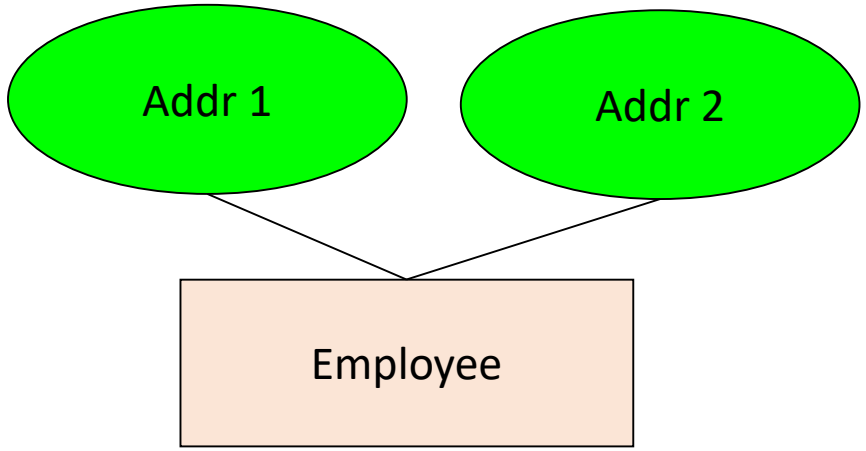


How do we handle employees with multiple addresses here?

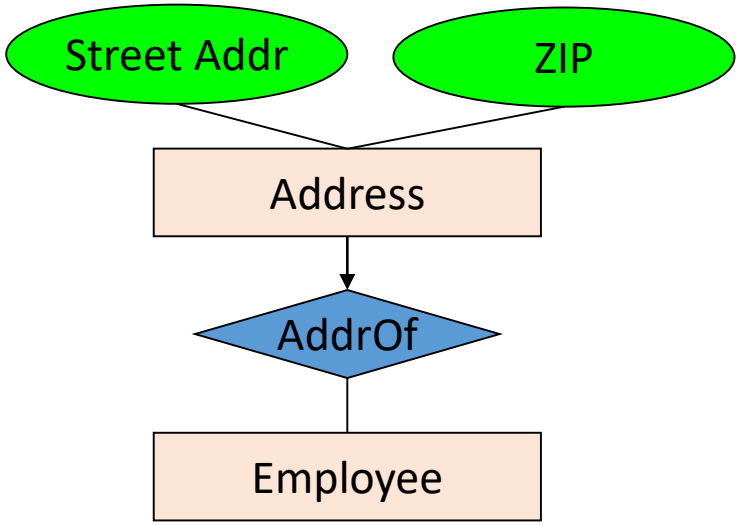
How do we handle addresses where internal structure of the address (e.g. zip code, state) is useful?

Examples: Entity vs. Attribute

Should address (A) be an attribute?



Or (B) be an entity?



In general, when we want to record several values, we choose new entity

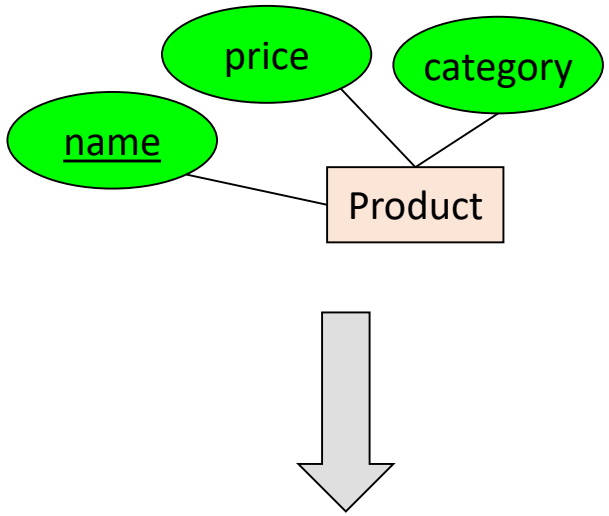
From E/R Diagrams to Relational Schema

- Key concept:

  Both ***Entity sets*** and ***Relationships*** become relations (tables in RDBMS)

From E/R Diagrams to Relational Schema

- An entity set becomes a relation (multiset of tuples / table)
  - Each tuple is one entity
  - Each tuple is composed of the entity's attributes, and has the same primary key

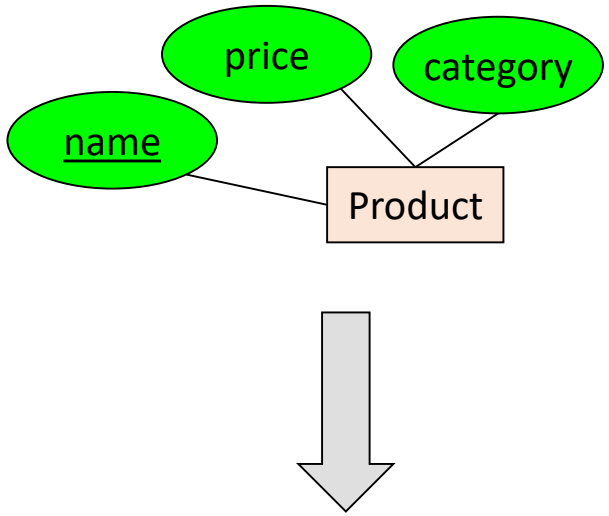


Product

name	price	category
Gizmo1	99.99	Camera
Gizmo2	19.99	Edible

From E/R Diagrams to Relational Schema

```
CREATE TABLE Product(
  name     CHAR(50) PRIMARY KEY,
  price    DOUBLE,
  category VARCHAR(30)
)
```

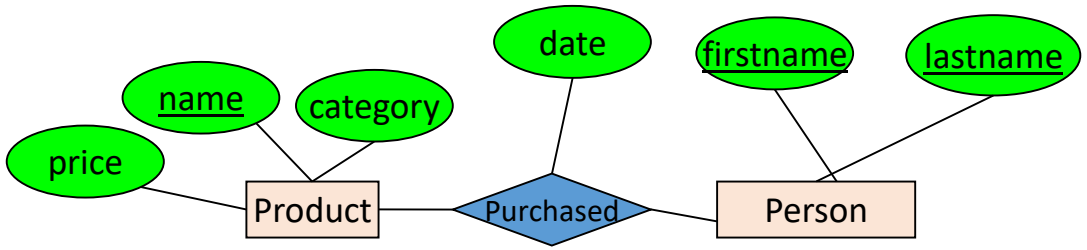


Product

name	price	category
Gizmo1	99.99	Camera
Gizmo2	19.99	Edible

From E/R Diagrams to Relational Schema

- A relation <u>between entity sets $A_1, \ldots, A_N$</u> *also* becomes a multiset of tuples / a table
  - Each row/tuple is one relation, i.e. one unique combination of entities $(a_1,\ldots,a_N)$
  - Each row/tuple is
    - composed of the **union of the entity sets' keys**
    - has the entities' primary keys as foreign keys
    - has the union of the entity sets' keys as primary key

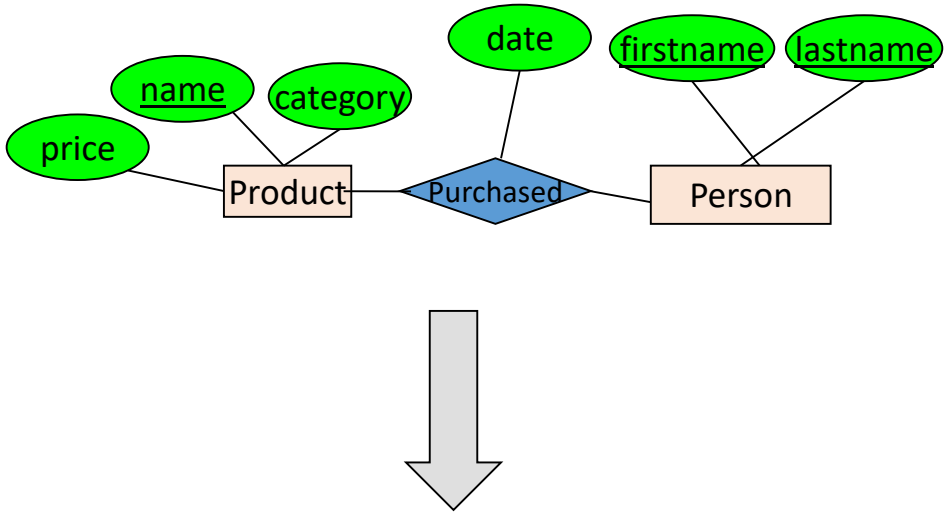


Purchased

name	firstname	lastname	date
Gizmo1	Bob	Joe	01/01/15
Gizmo2	Joe	Bob	01/03/15
Gizmo1	JoeBob	Smith	01/05/15

From E/R Diagrams to Relational Schema

```
CREATE TABLE Purchased(
  name      CHAR(50),
  firstname CHAR(50),
  lastname  CHAR(50),
  date      DATE,
  PRIMARY KEY (name, firstname, lastname),
  FOREIGN KEY (name)
          REFERENCES Product,
  FOREIGN KEY (firstname, lastname)
          REFERENCES Person
)
```

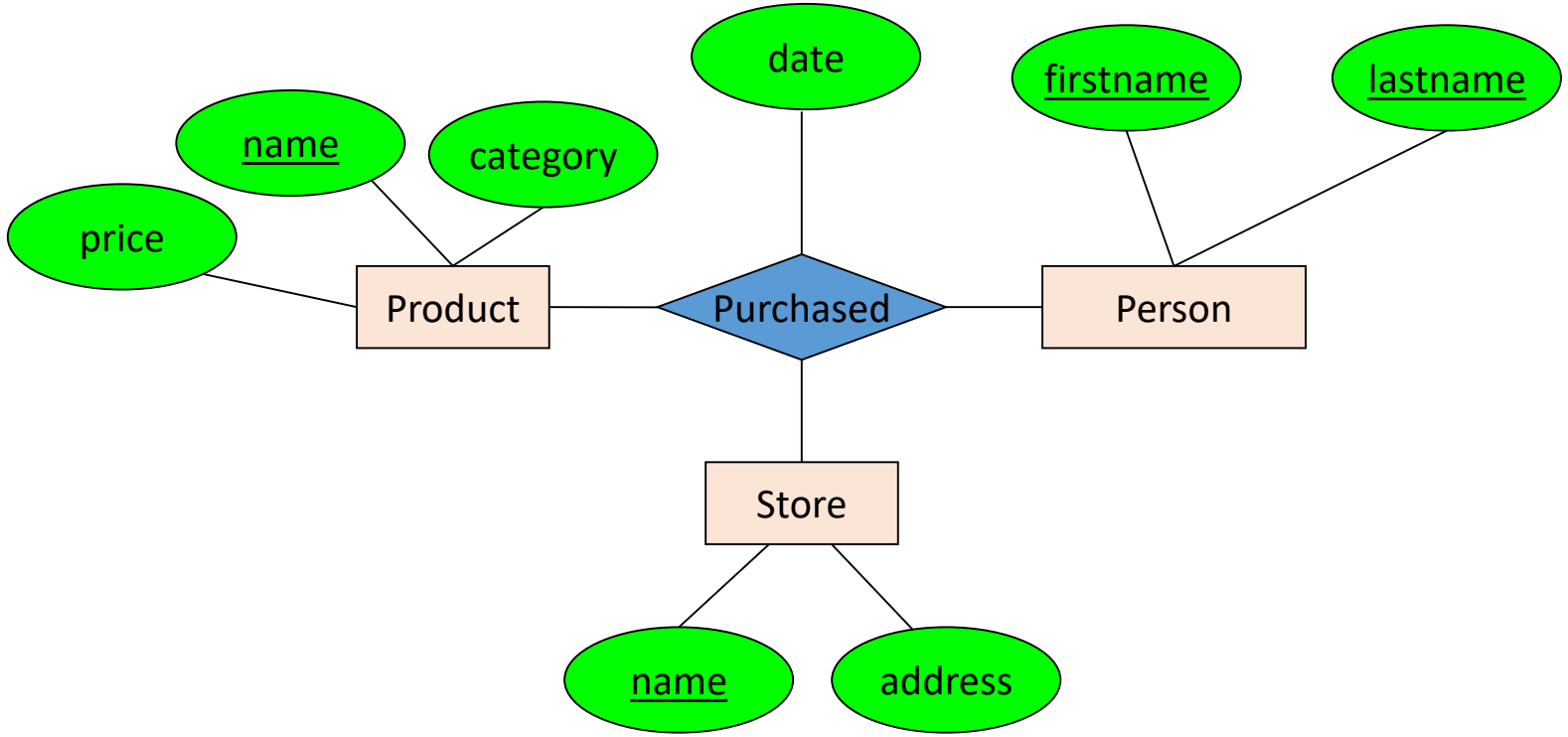


Purchased

name	firstname	lastname	date
Gizmo1	Bob	Joe	01/01/15
Gizmo2	Joe	Bob	01/03/15
Gizmo1	JoeBob	Smith	01/05/15

From E/R Diagram to Relational Schema

How do we represent this as a relational schema?



Add arrows to your E/R diagram!

Also make sure to add **(new concepts underlined):**

A player can only belong to one team, a play can only be in one game, a pass/run..?

Players can achieve a **Personal Record** linked to a specific Game and Play

Players have a **weight** which changes in on vs. off-season

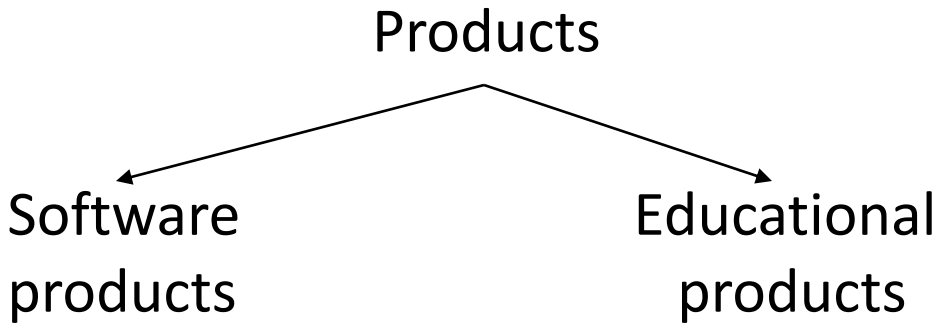
3. Advanced E/R Concepts

What you will learn about in this section

1. Subclasses & connection to OO
2. Constraints
3. Weak entity sets

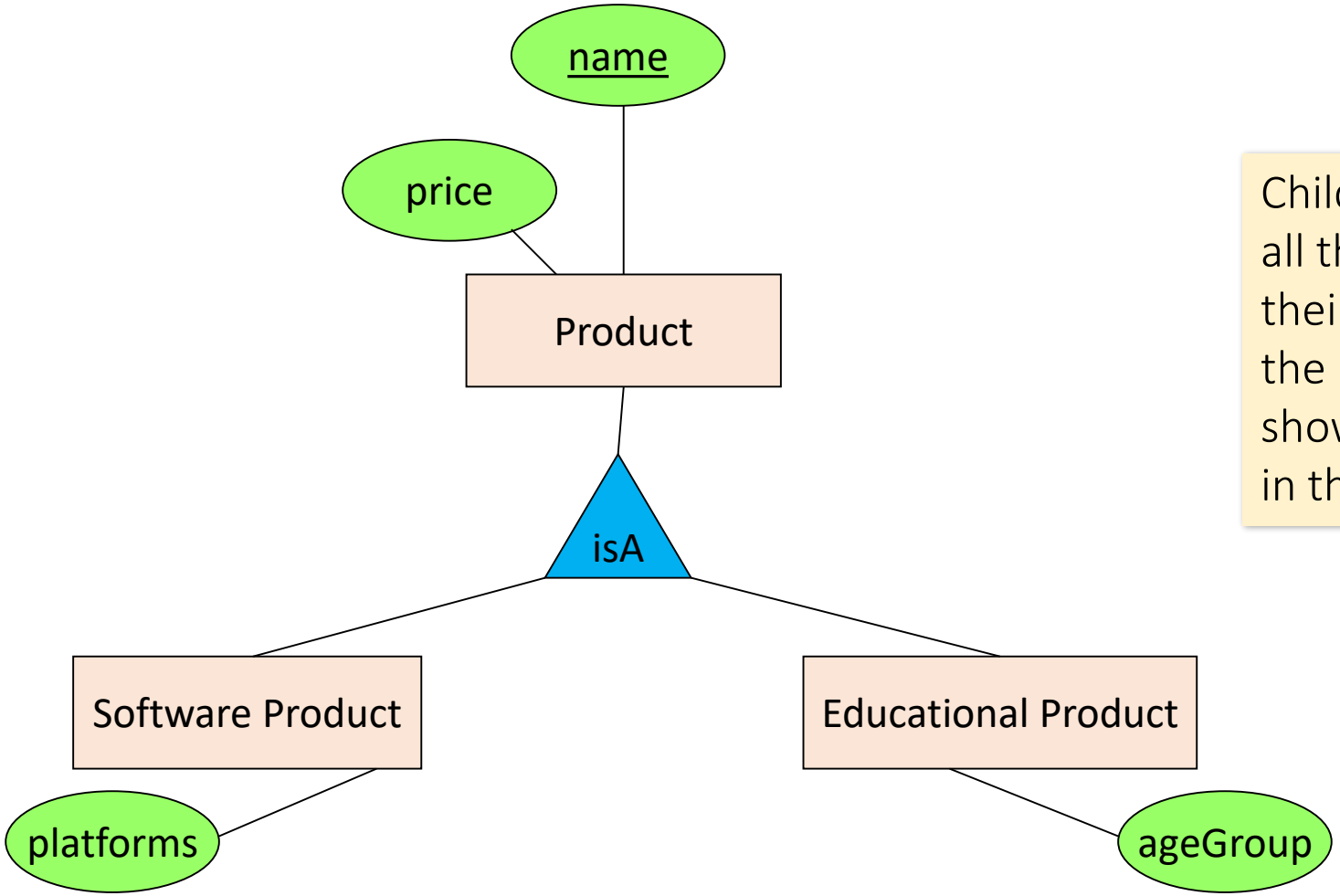
Modeling Subclasses

- Some objects in a class may be special, i.e. worthy of their own class
  - Define a new class?
    - *But what if we want to maintain connection to current class?*
  - <u>Better: define a *subclass*</u>
    - *Ex:*



We can define **subclasses** in E/R!

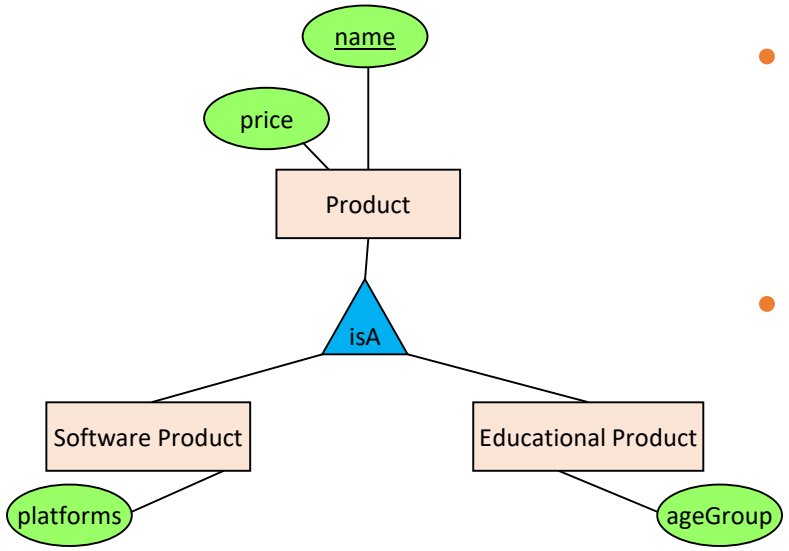
Modeling Subclasses



Child subclasses contain all the attributes of *all* of their parent classes **plus** the new attributes shown attached to them in the E/R diagram

Understanding Subclasses

- Think in terms of records; ex:



- Product

name
price

- SoftwareProduct

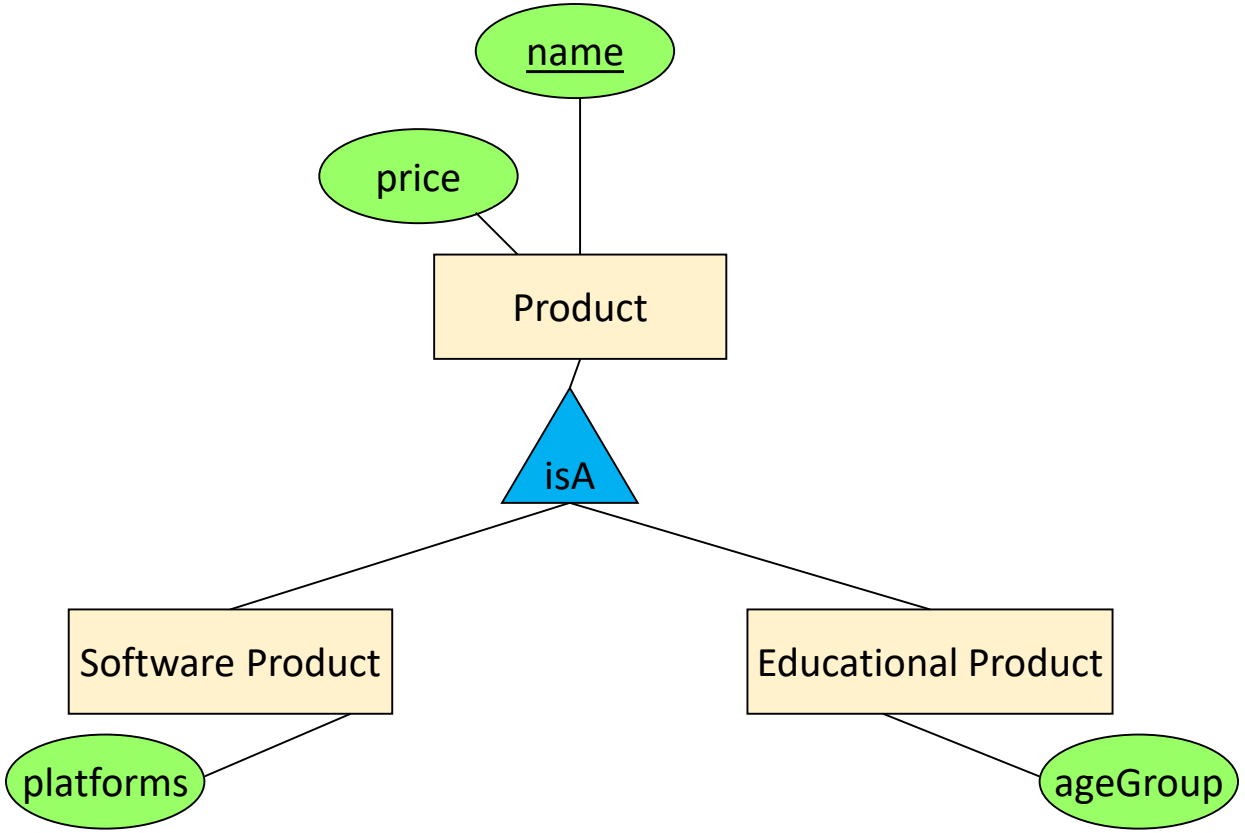
name
price
platforms

- EducationalProduct

name
price
ageGroup

Child subclasses contain all the attributes of *all* of their parent classes **plus** the new attributes shown attached to them in the E/R diagram

Think like tables…



Product

name	price	category
Gizmo	99	gadget
Camera	49	photo
Toy	39	gadget

Sw.Product

name	platforms
Gizmo	unix

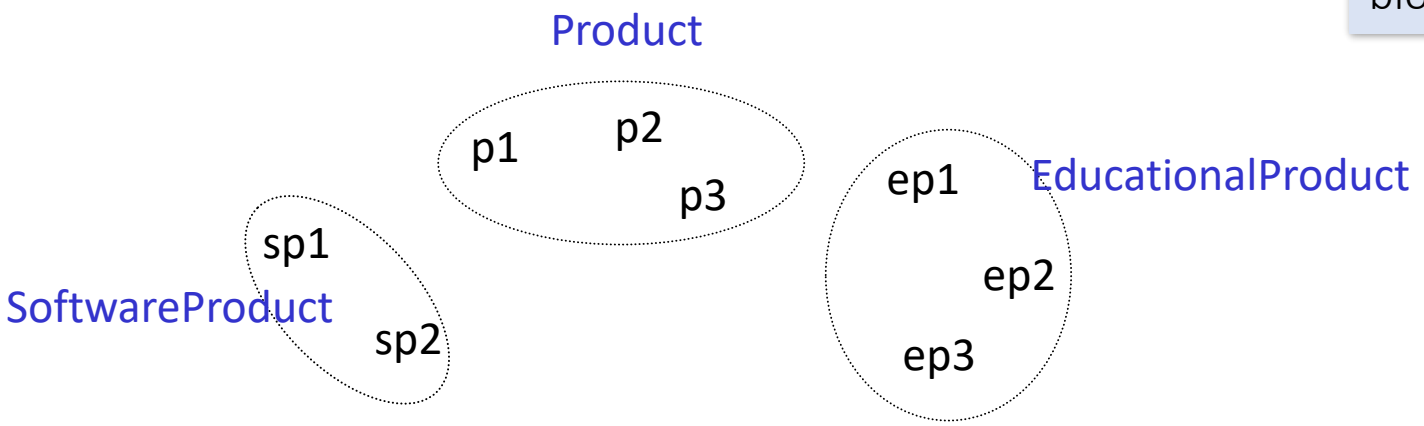
Ed.Product

name	ageGroup
Gizmo	todler
Toy	retired

Difference between OO and E/R inheritance

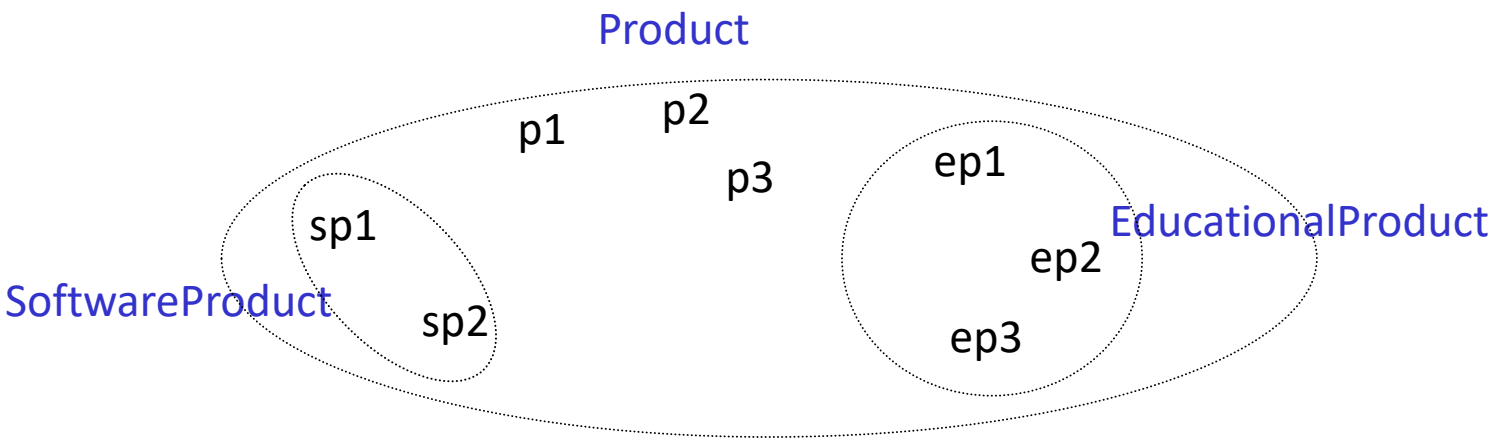
- OO: Classes are disjoint (same for Java, C++)

OO = <u>Object Oriented</u>. E.g. classes as fundamental building block, etc…



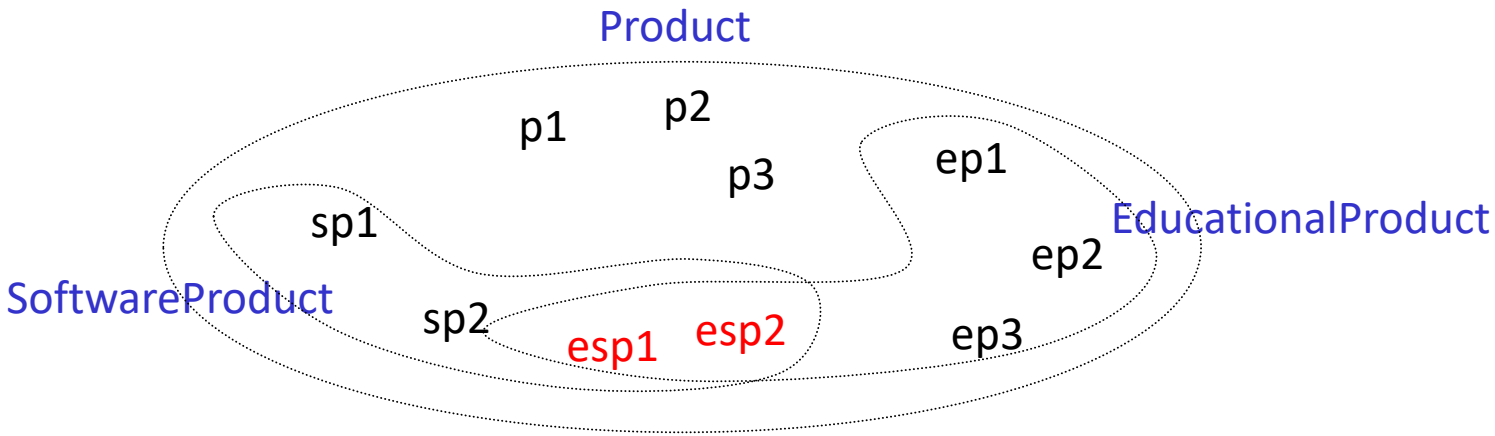
Difference between OO and E/R inheritance

- E/R: entity sets overlap



Difference between OO and E/R inheritance

We have three entity sets, but four different kinds of objects



No need for multiple inheritance in E/R

IsA Review

- If we declare ***A IsA B*** then every **A** is a **B**
- We use IsA to
  - Add descriptive attributes to a subclass
  - To identify entities that participate in a relationship
- **No need for multiple inheritance**

Modeling UnionTypes With Subclasses

Person

FurniturePiece

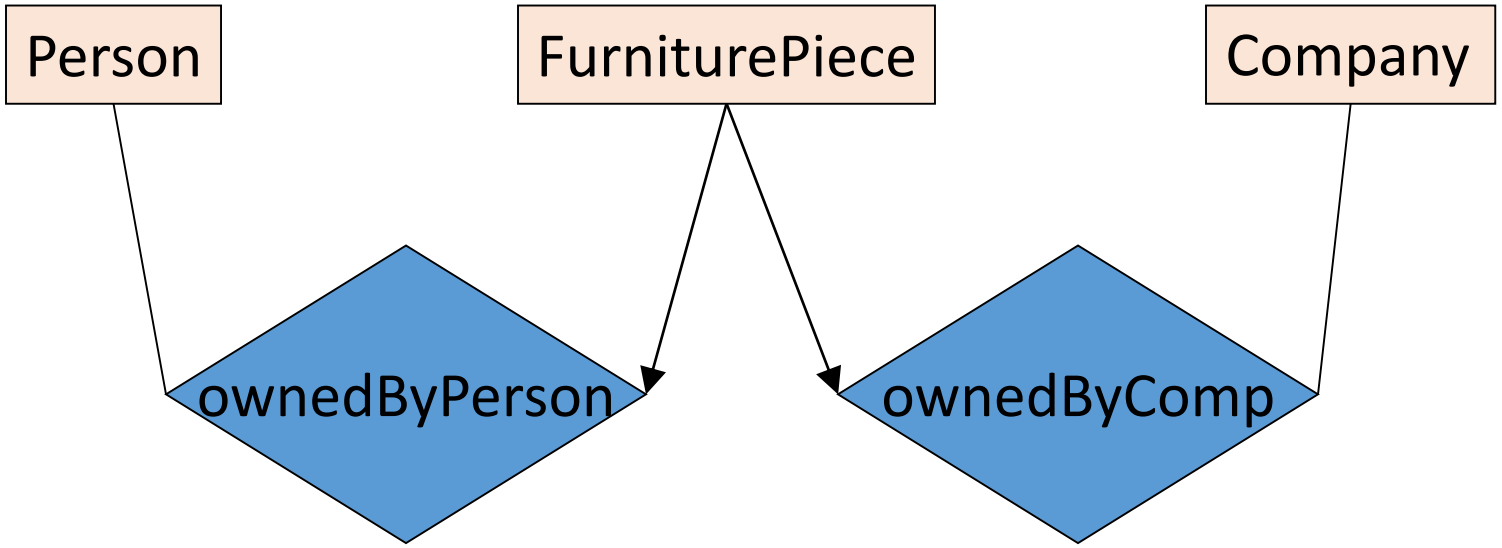
Company

Suppose each piece of furniture is owned either by a person, or by a company. *How do we represent this?*

Modeling Union Types with Subclasses

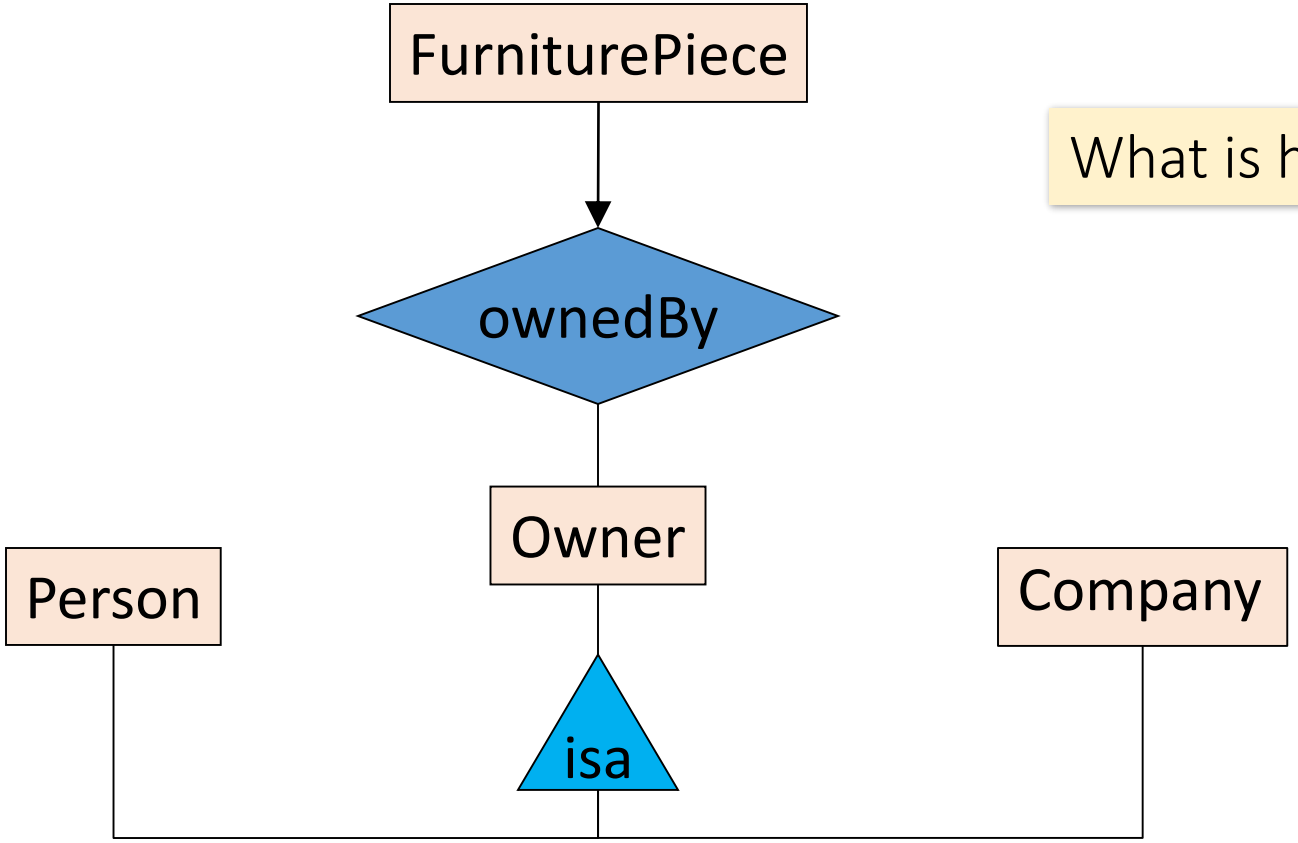
Say: each piece of furniture is owned either by a person, or by a company

<u>**Solution 1.**</u> Acceptable, but imperfect (What's wrong ?)



Modeling Union Types with Subclasses

Solution 2: better (though more laborious)



What is happening here?

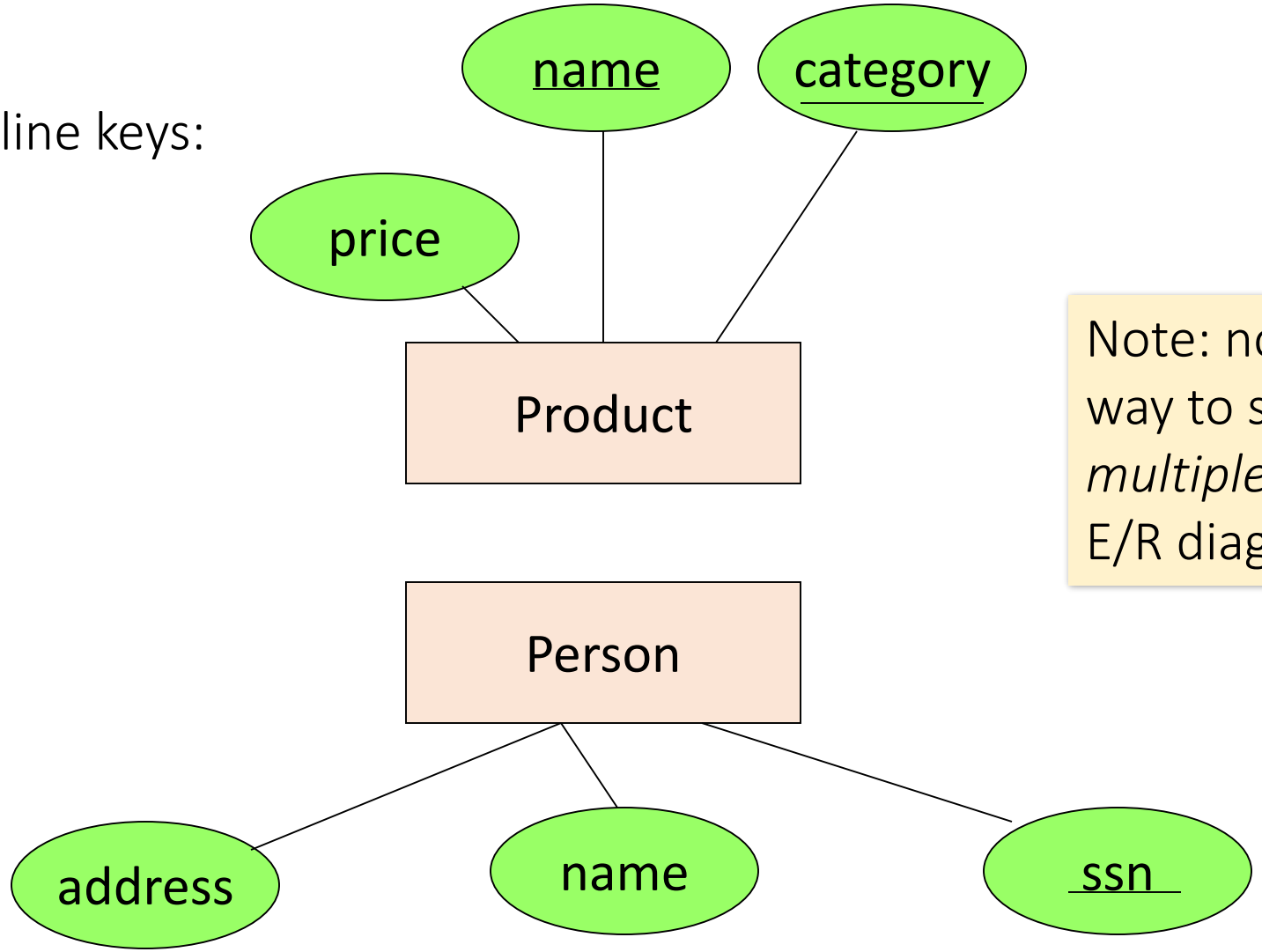
Constraints in E/R Diagrams

- Finding constraints is part of the E/R modeling process. Commonly used constraints are:
  - Keys: Implicit constraints on uniqueness of entities
    - *Ex: An SSN uniquely identifies a person*
  - Single-value constraints:
    - *Ex: a person can have only one father*
  - Referential integrity constraints: Referenced entities must exist
    - *Ex: if you work for a company, it must exist in the database*
  - Other constraints:
    - *Ex: peoples' ages are between 0 and 150*

Recall FOREIGN KEYs!

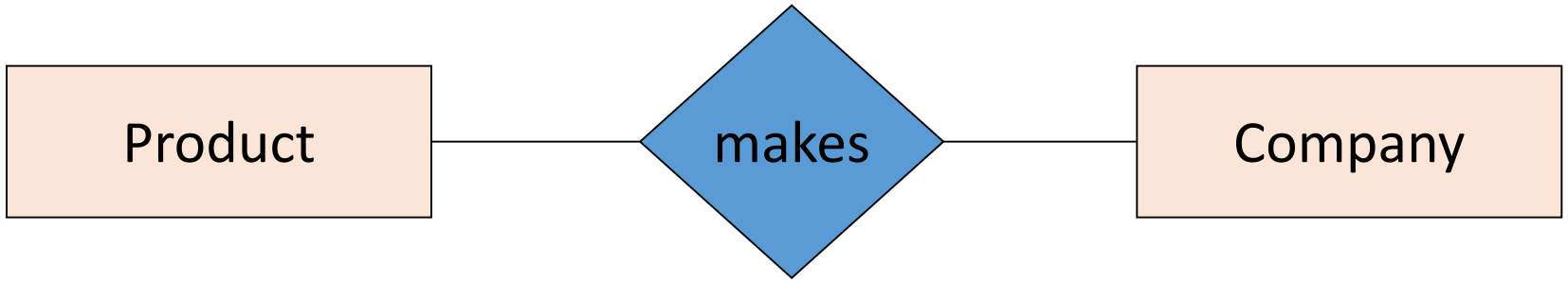
Keys in E/R Diagrams

Underline keys:

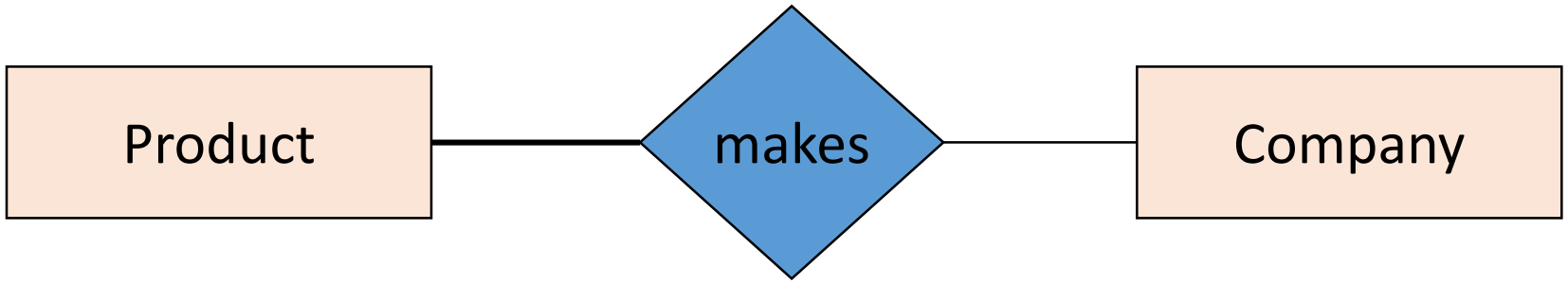


Note: no formal way to specify *multiple* keys in E/R diagrams…

Participation Constraints: Partial v. Total

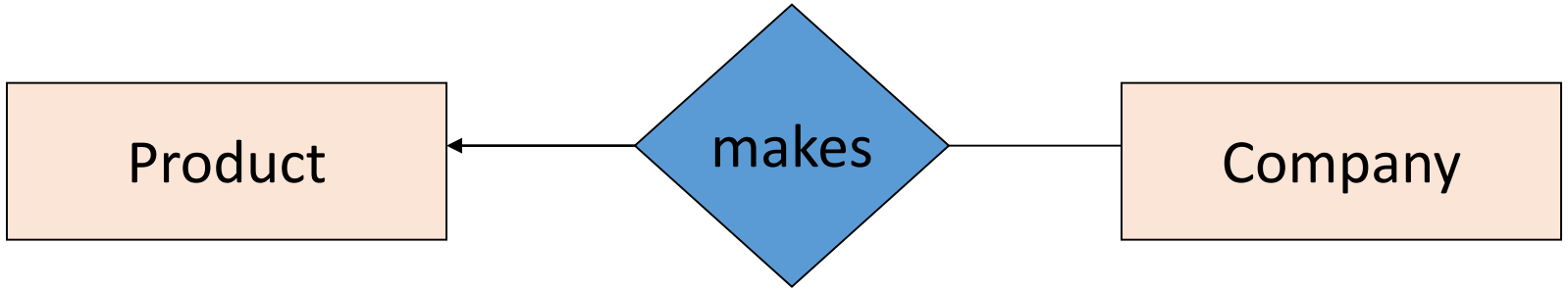


Are there products made by no company?
Companies that don't make a product?

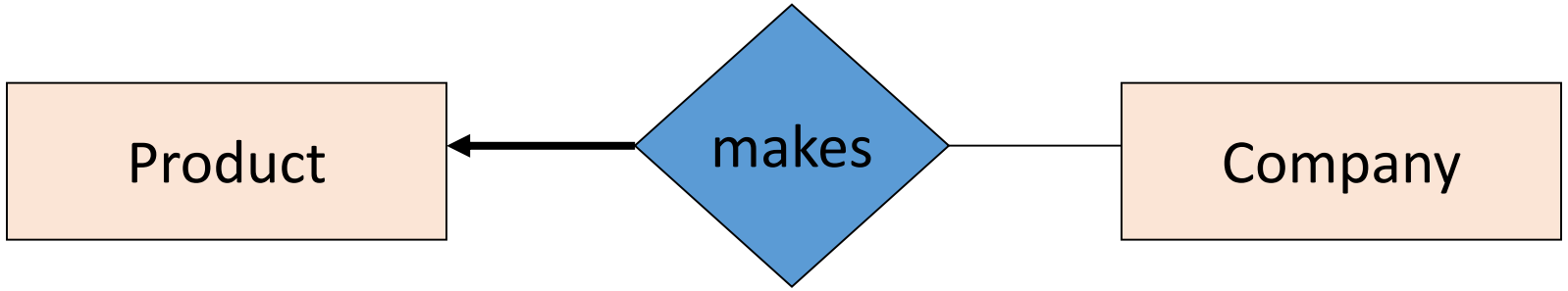


Bold line indicates *total participation* (i.e. here: all products are made by a company)

Referential Integrity Constraints



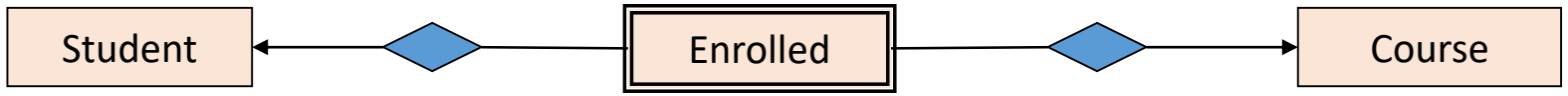
Each product made by at most one company.
Some products made by no company?



Each product made by *exactly* one company.

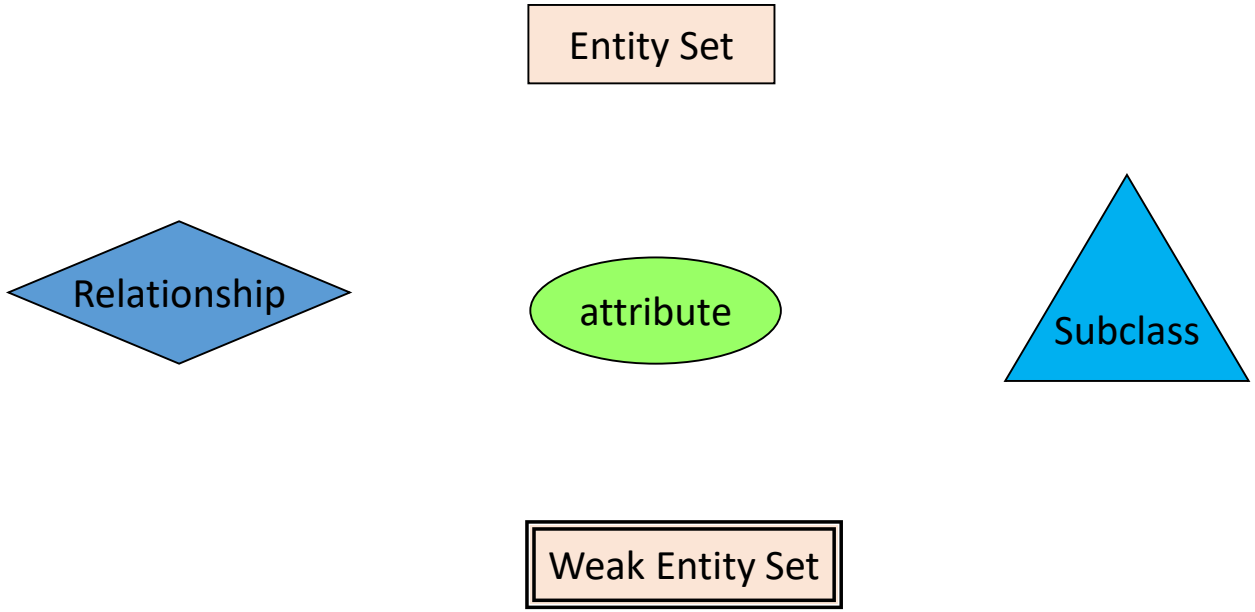
Weak Entity Sets

Entity sets are *weak* when their key comes from other classes to which they are related.

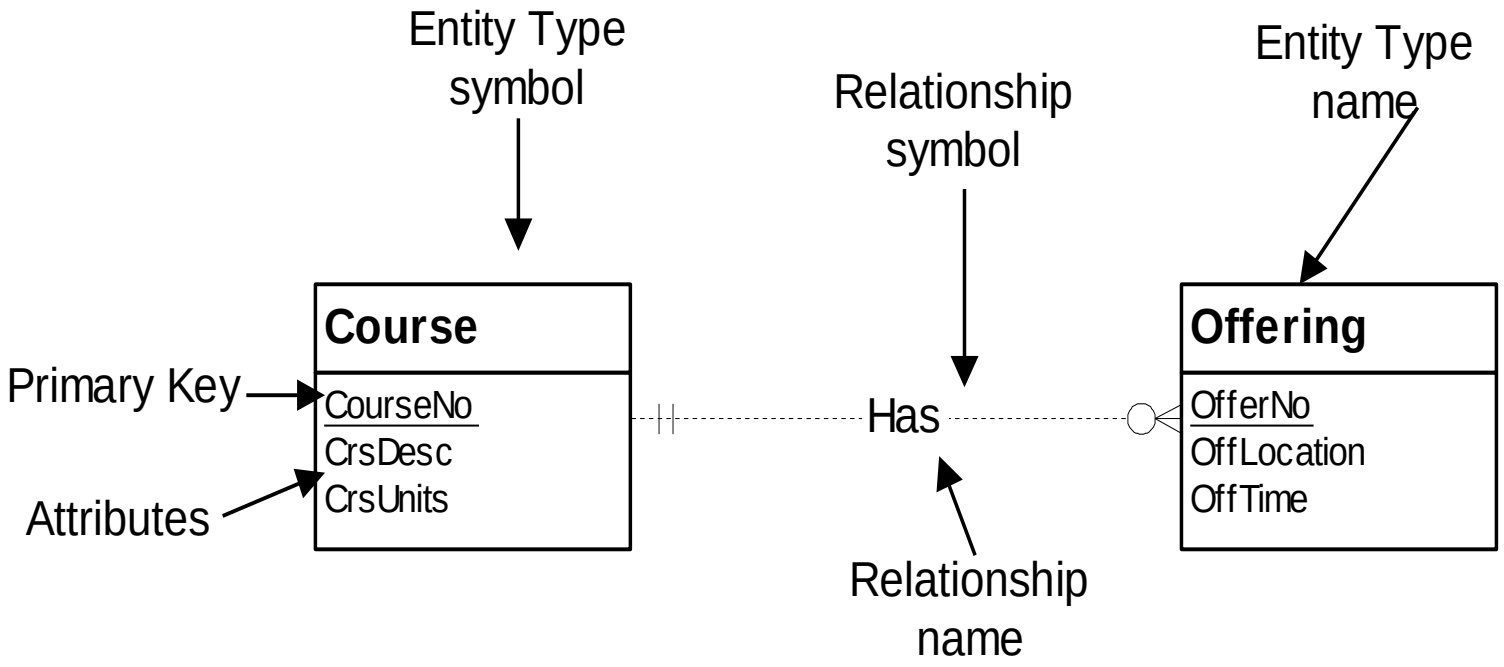


Equal to:

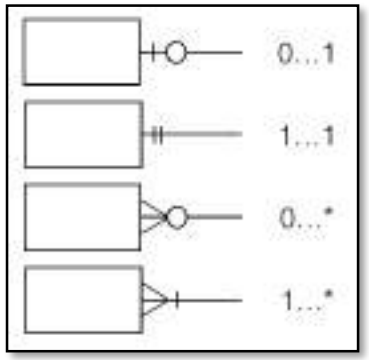
Summary of Used Symbols

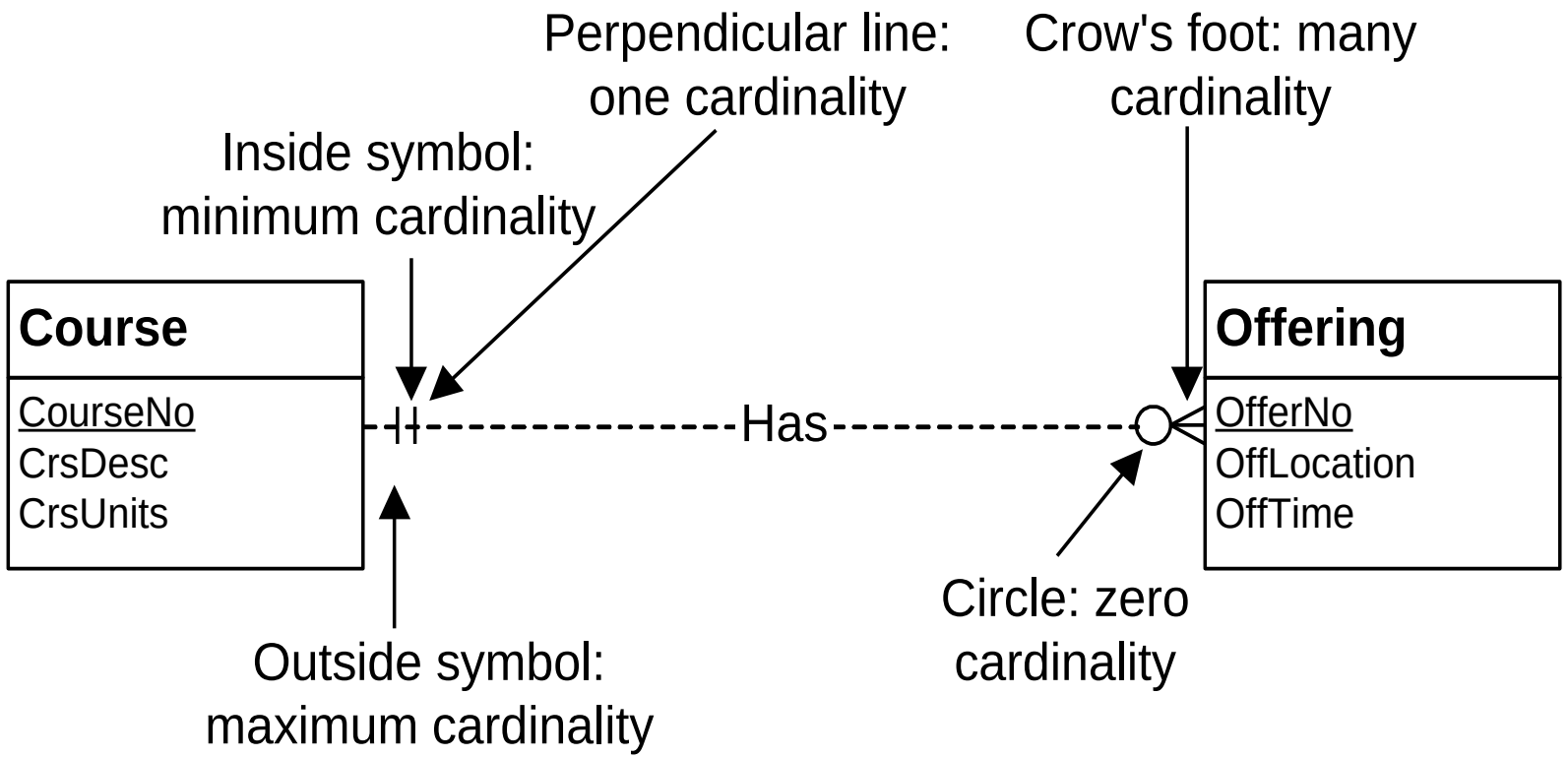


Alternative Representations: Basic Symbols

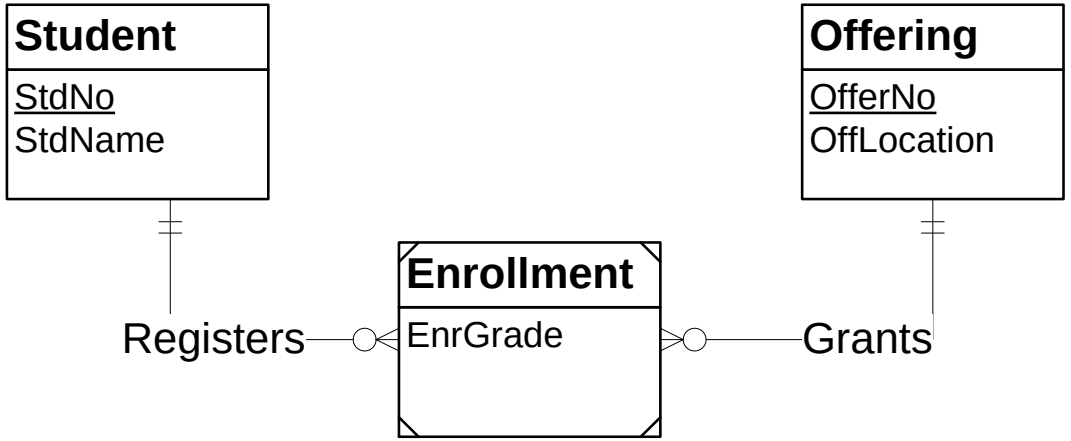


Alternative Representations: Cardinality

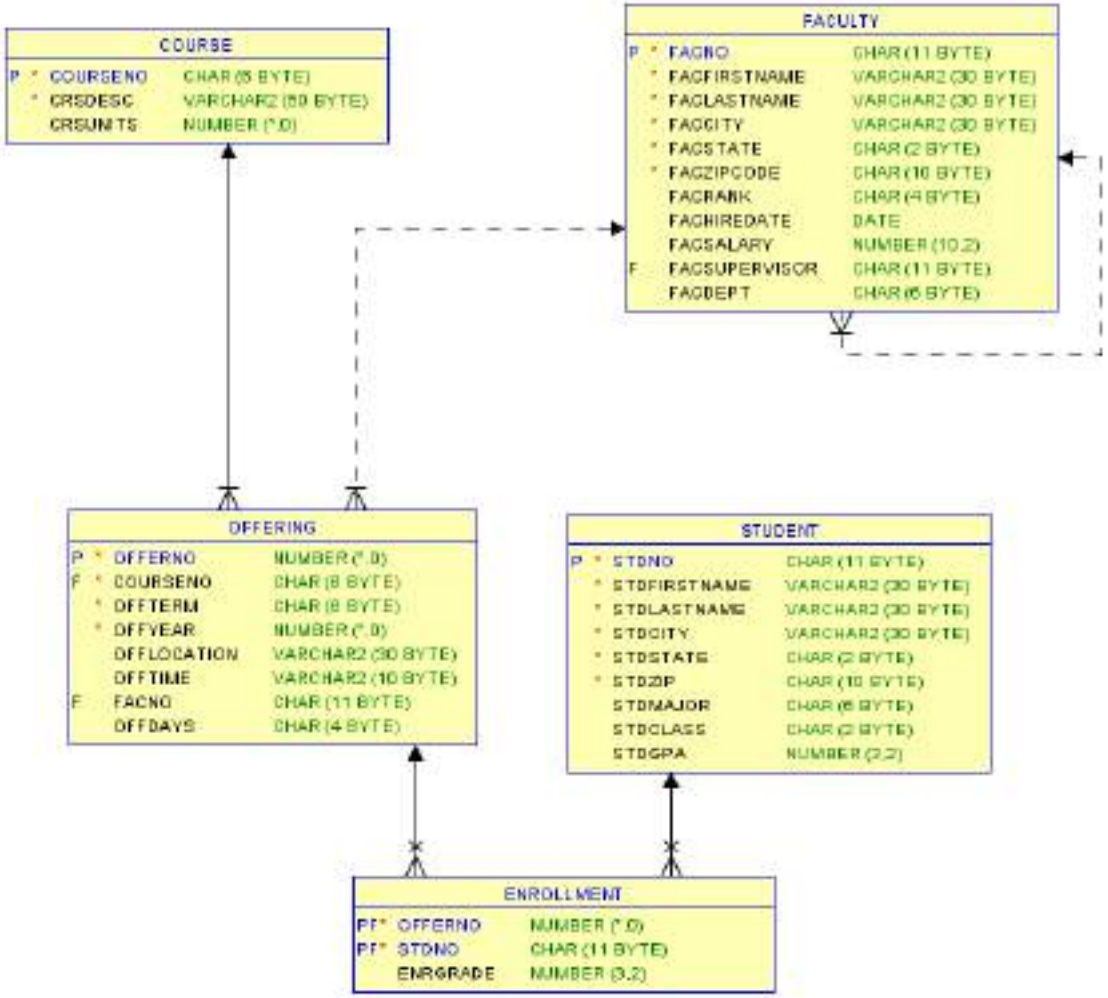




Alternative Representations: Example



Alternative Representations: Tool X



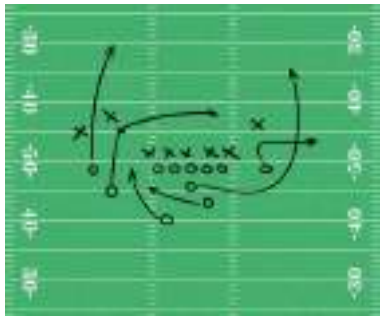
Add in: Subclasses, constraints, and weak entity sets

Concepts to include / model:

Teams belong to cities- model as ***weak entity sets***

Players are either on Offense or Defense, and are of types (QB, RB, WR, TE, K, Farmer*…)

All passes are to exactly one player; all runs include a player



Make sure you have designated keys for all our concepts!

*https://twitter.com/McBPJ/status/638728908628586496/photo/1

E/R Summary

- E/R diagrams are a visual syntax that allows technical and non-technical people to talk
  - For conceptual design
- Basic constructs: **entity**, **relationship**, and **attributes**
- A good design is faithful to the constraints of the application, but not overzealous

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the *CS145 Introduction to Databases* lecture given by *Christopher Ré* at *Stanford University* (http://web.stanford.edu/class/cs145/).

Entity-Relationship Model

Entity-Relationship Model

- Design Process
- Modeling
- Constraints
- E-R Diagram
- Design Issues
- Weak Entity Sets
- Extended E-R Features
- Design of the Bank Database
- Reduction to Relation Schemas
- Database Design
- UML

Design Phases

- The initial phase of database design is to characterize fully the data needs of the prospective database users.
- Next, the designer chooses a data model and, by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database.
- A fully developed conceptual schema also indicates the functional requirements of the enterprise. In a “specification of functional requirements”, users describe the kinds of operations (or transactions) that will be performed on the data.

Design Phases (Cont.)

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.

- Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.
  - Business decision – What attributes should we record in the database?
  - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database

Design Approaches

- Entity Relationship Model (covered in this chapter)
  - Models an enterprise as a collection of *entities* and *relationships*
    - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
      - Described by a set of *attributes*
    - Relationship: an association among several entities
  - Represented diagrammatically by an *entity-relationship diagram:*
- Normalization Theory (Chapter 8)
  - Formalize what designs are bad, and test for them

Outline of the ER Model

ER model -- Database Modeling

- The ER data mode was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database.
- The ER model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the ER model.
- The ER data model employs three basic concepts:
  - entity sets,
  - relationship sets,
  - attributes.
- The ER model also has an associated diagrammatic representation, the ER diagram, which can express the overall logical structure of a database graphically.

Entity Sets

- An **entity** is an object that exists and is distinguishable from other objects.
  - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the same type that share the same properties.
  - Example: set of all persons, companies, trees, holidays
- An entity is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.
  - Example:

    instructor = (*ID, name, street, city, salary*)
    course= (*course_id, title, credits*)
- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifiying each member of the set.

Entity Sets -- *instructor* and *student*

instructor_ID	instructor_name
76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

student-ID	student_name
98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Relationship Sets

- A **relationship** is an association among several entities

  Example:

44553 (Peltier)	*advisor*	22222 (Einstein)
student entity	relationship set	*instructor* entity

- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

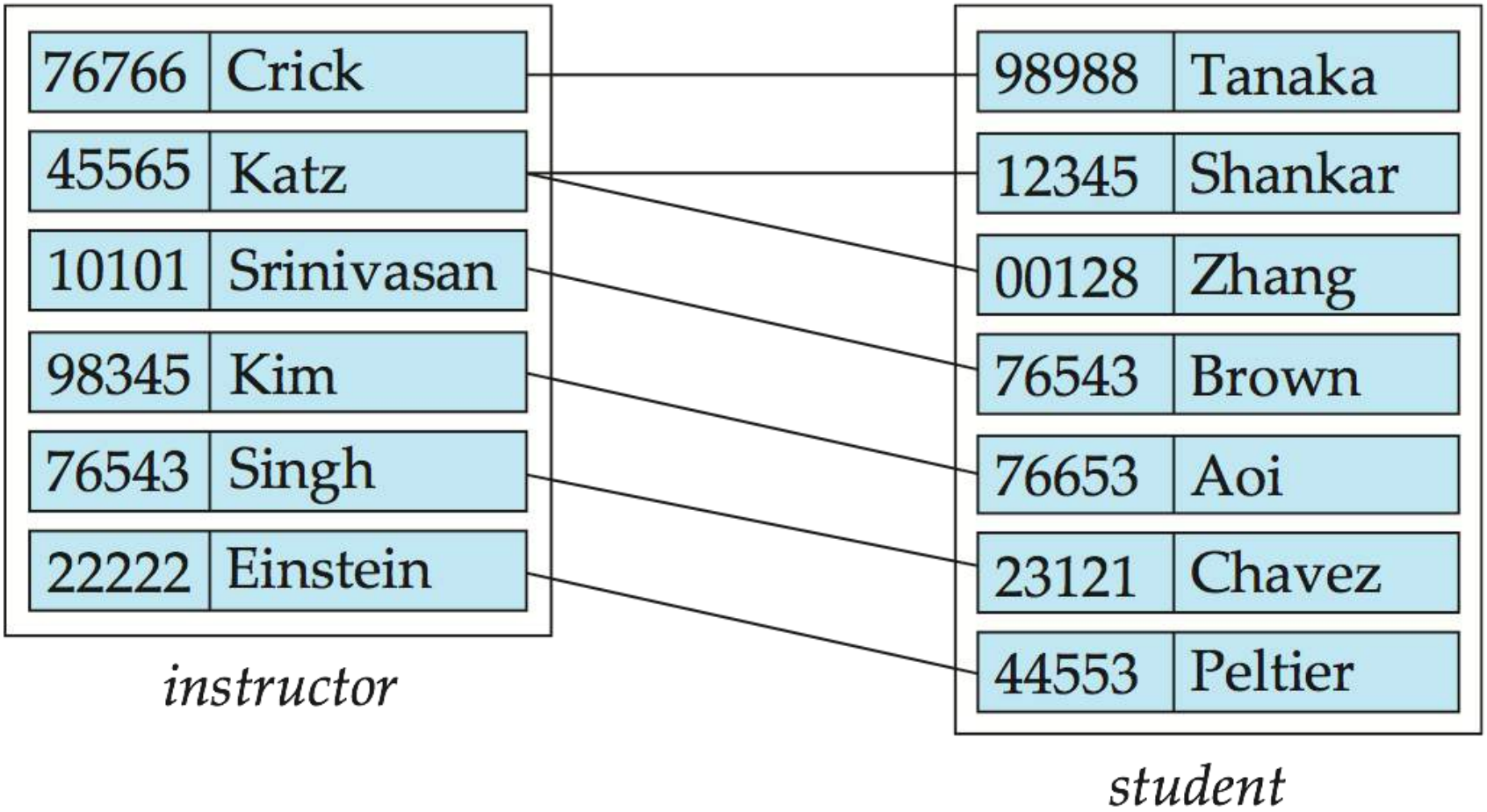
$$\{(e_1, e_2, \ldots e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

  where $(e_1, e_2, \ldots, e_n)$ is a relationship

  - Example:

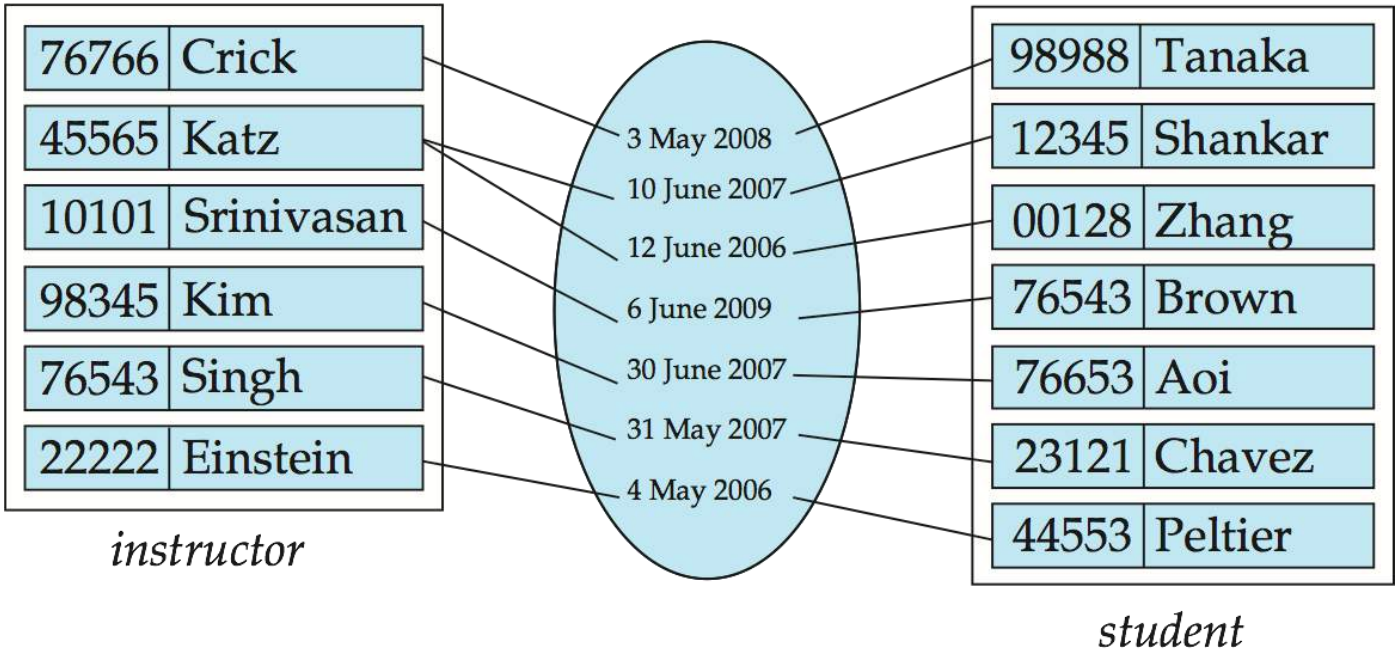
$$(44553,22222) \in advisor$$

Relationship Set *advisor*



Relationship Sets (Cont.)

- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor



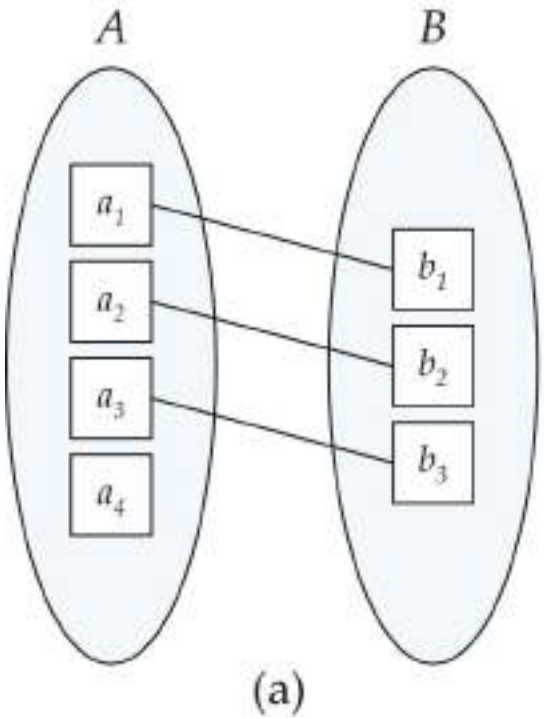
Degree of a Relationship Set

- binary relationship
  - involve two entity sets (or degree two).
  - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
  - Example: *students* work on research *projects* under the guidance of an *instructor*.
  - relationship *proj_guide* is a ternary relationship between *instructor, student,* and *project*

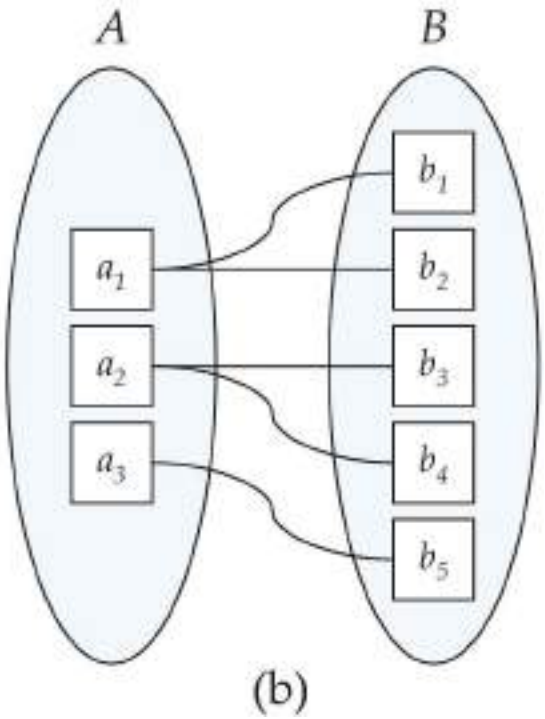
Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one
  - Many to many

Mapping Cardinalities



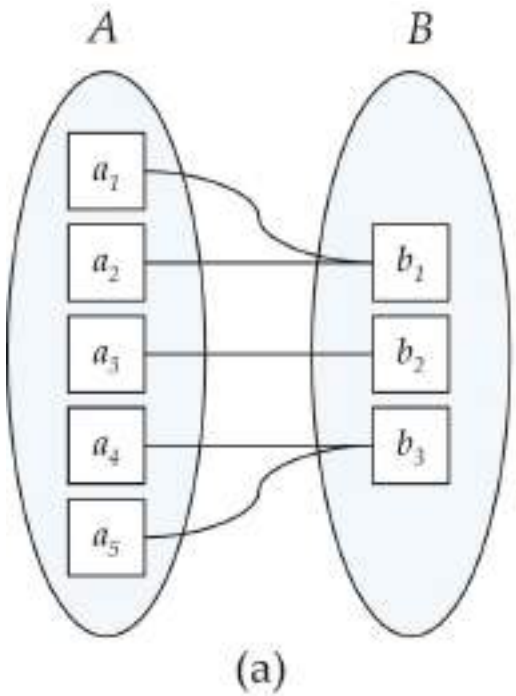
One to one



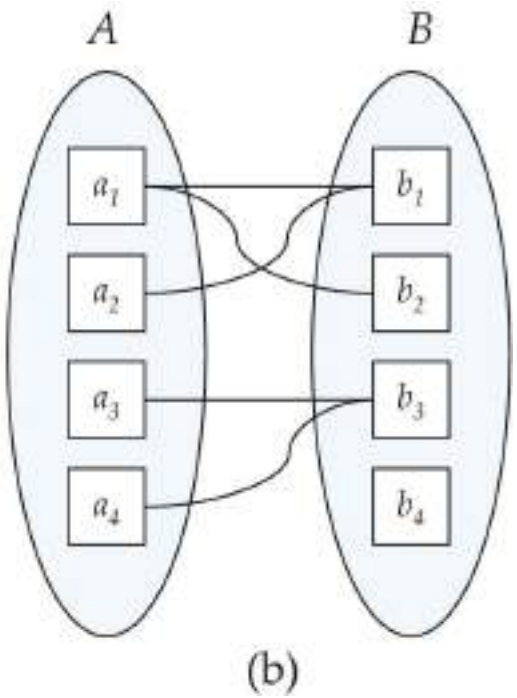
One to many

Note: Some elements in *A* and *B* may not be mapped to any elements in the other set

Mapping Cardinalities



Many to one



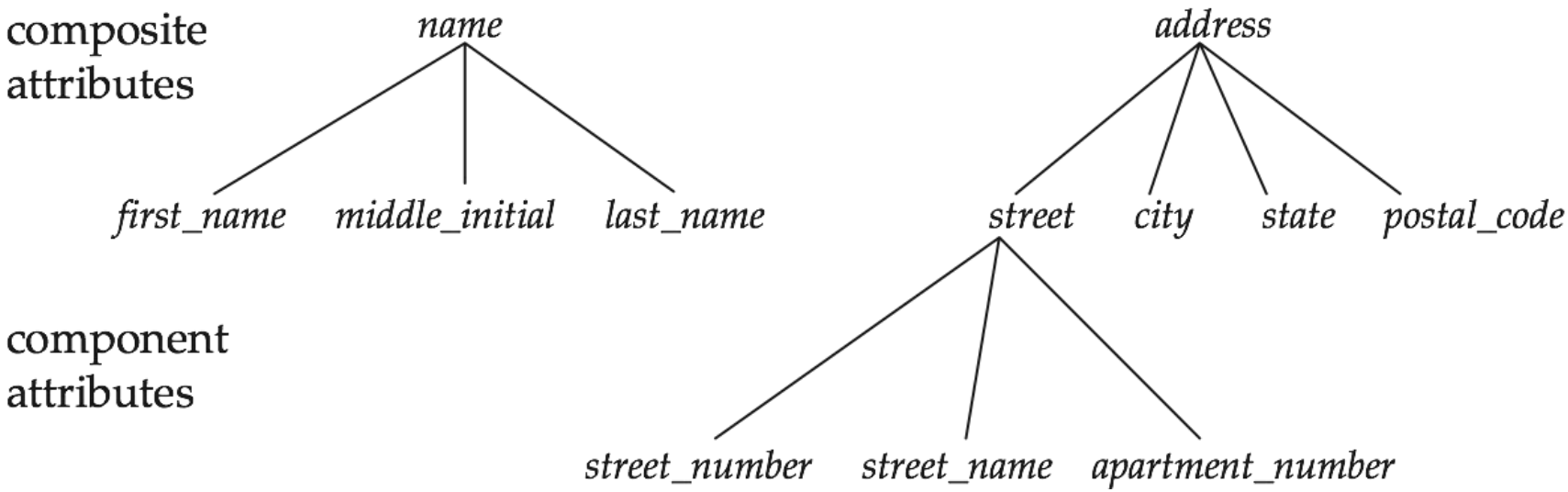
Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set

Complex Attributes

- Attribute types:
  - **Simple** and **composite** attributes.
  - **Single-valued** and **multivalued** attributes
    - Example: multivalued attribute: *phone_numbers*
  - **Derived** attributes
    - Can be computed from other attributes
    - Example: age, given date_of_birth
- **Domain** – the set of permitted values for each attribute

Composite Attributes



Redundant Attributes

- Suppose we have entity sets:
  - *instructor*, with attributes: *ID*, *name*, *dept_name, salary*
  - *department,* with attributes: *dept_name, building, budget*
- We model the fact that each instructor has an associated department using a relationship set *inst_dept*
- The attribute *dept_name* appears in both entity sets. Since it is the primary key for the entity set *department*, it replicates information present in the relationship and is therefore redundant in the entity set *instructor* and needs to be removed.
- BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see later.

Weak Entity Sets

- Consider a *section* entity, which is uniquely identified by a *course_id*, *semester, year*, and *sec_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.
- Note that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship *sec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.

Weak Entity Sets (Cont.)

- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester.* However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely; although each *section* entity is distinct, sections for different courses may share the same *section_id*, *year*, and *semester*.
- To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case, the *course_id*, required to identify *section* entities uniquely.
- The notion of **weak entity set** formalizes the above intuition. A weak entity set is one whose existence is dependent on another entity, called its **identifying entity**; instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity. An entity set that is not a weak entity set is termed a **strong entity set.**

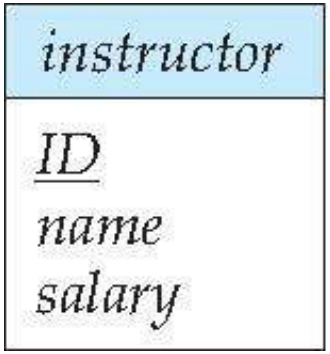
Weak Entity Sets (Cont.)

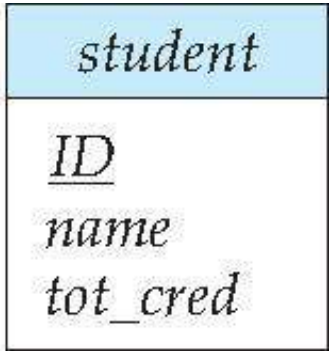
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section.*

E-R Diagrams

Entity Sets

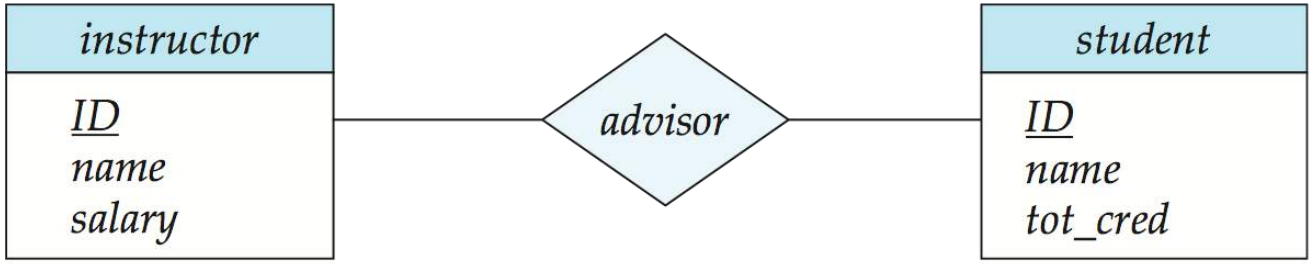
- Entities can be represented graphically as follows:
  - Rectangles represent entity sets.
  - Attributes listed inside entity rectangle
  - Underline indicates primary key attributes



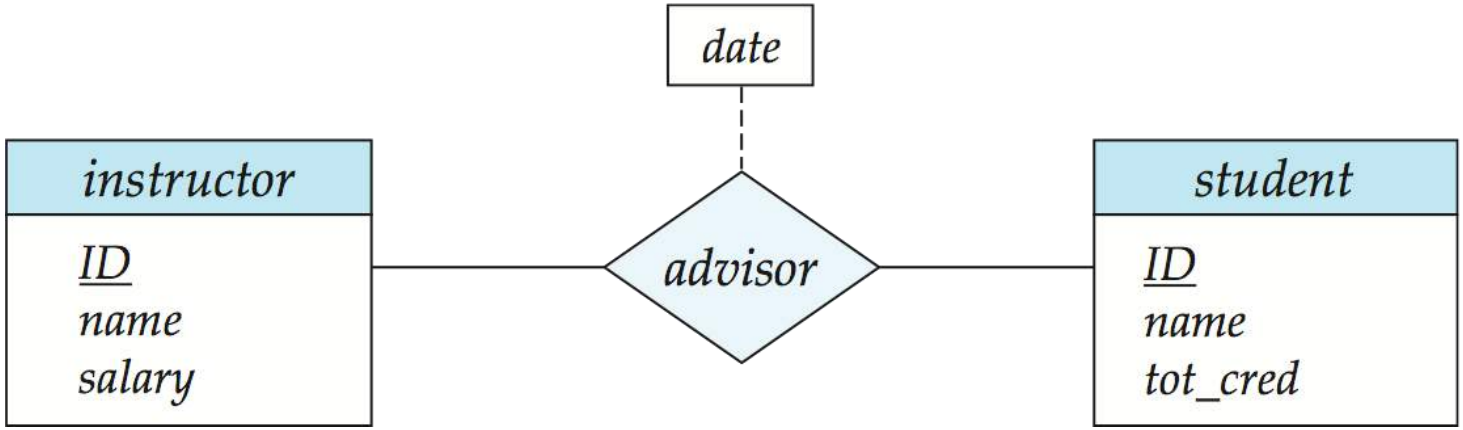


Relationship Sets

- Diamonds represent relationship sets.

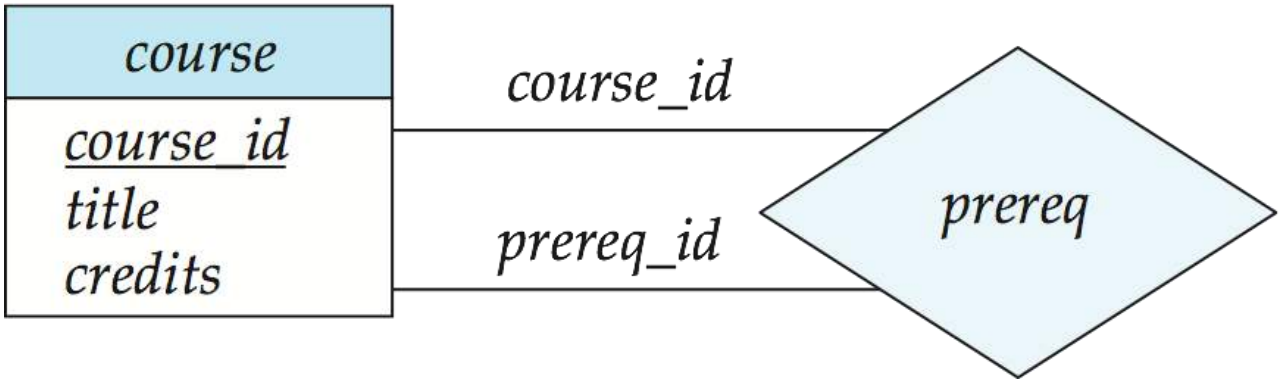


Relationship Sets with Attributes



Roles

- Entity sets of a relationship need not be distinct
  - Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.



Cardinality Constraints

- We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (—), signifying "many," between the relationship set and the entity set.

- One-to-one relationship between an *instructor* and a *student* :
  - A student is associated with at most one *instructor* via the relationship *advisor*
  - A *student* is associated with at most one *department* via *stud_dept*

One-to-Many Relationship

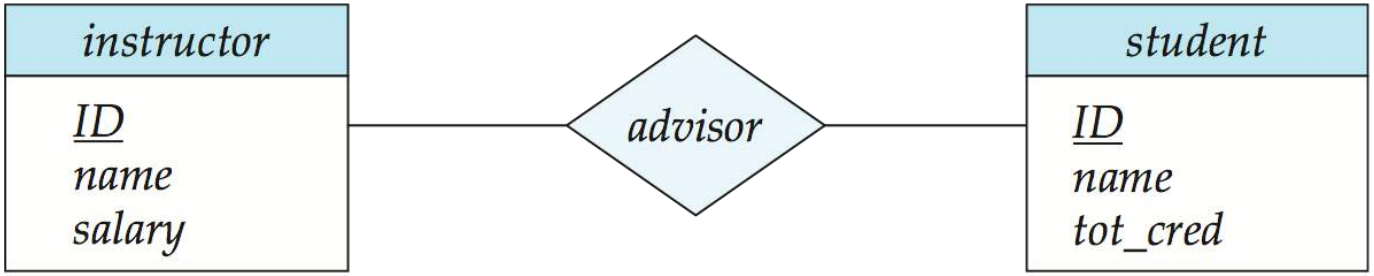
- one-to-many relationship between an *instructor* and a *student*
  - an instructor is associated with several (including 0) students via *advisor*
  - a student is associated with at most one instructor via advisor,

Many-to-One Relationships

- In a many-to-one relationship between an *instructor* and a *student,*
  - an instructor is associated with at most one student via *advisor*,
  - and a student is associated with several (including 0) instructors via *advisor*

Many-to-Many Relationship

- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*



Total and Partial Participation

- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set

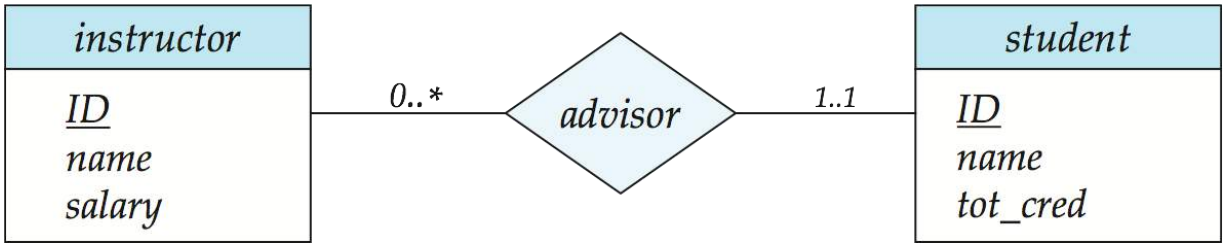
participation of *student* in *advisor* relation is total

  - every *student* must have an associated instructor

- Partial participation: some entities may not participate in any relationship in the relationship set
  - Example: participation of *instructor* in *advisor* is partial

Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form *l..h*, where *l* is the minimum and *h* the maximum cardinality
  - A minimum value of 1 indicates total participation.
  - A maximum value of 1 indicates that the entity participates in at most one relationship
  - A maximum value of * indicates no limit.



Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

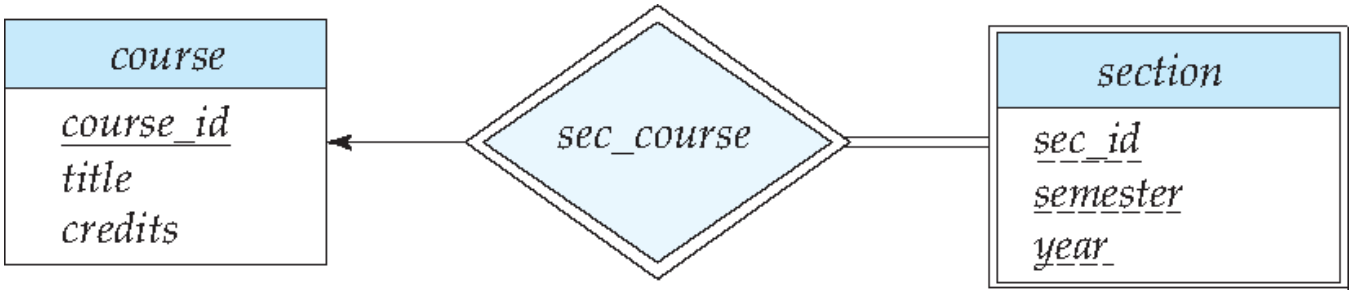
Notation to Express Entity with Complex Attributes

instructor

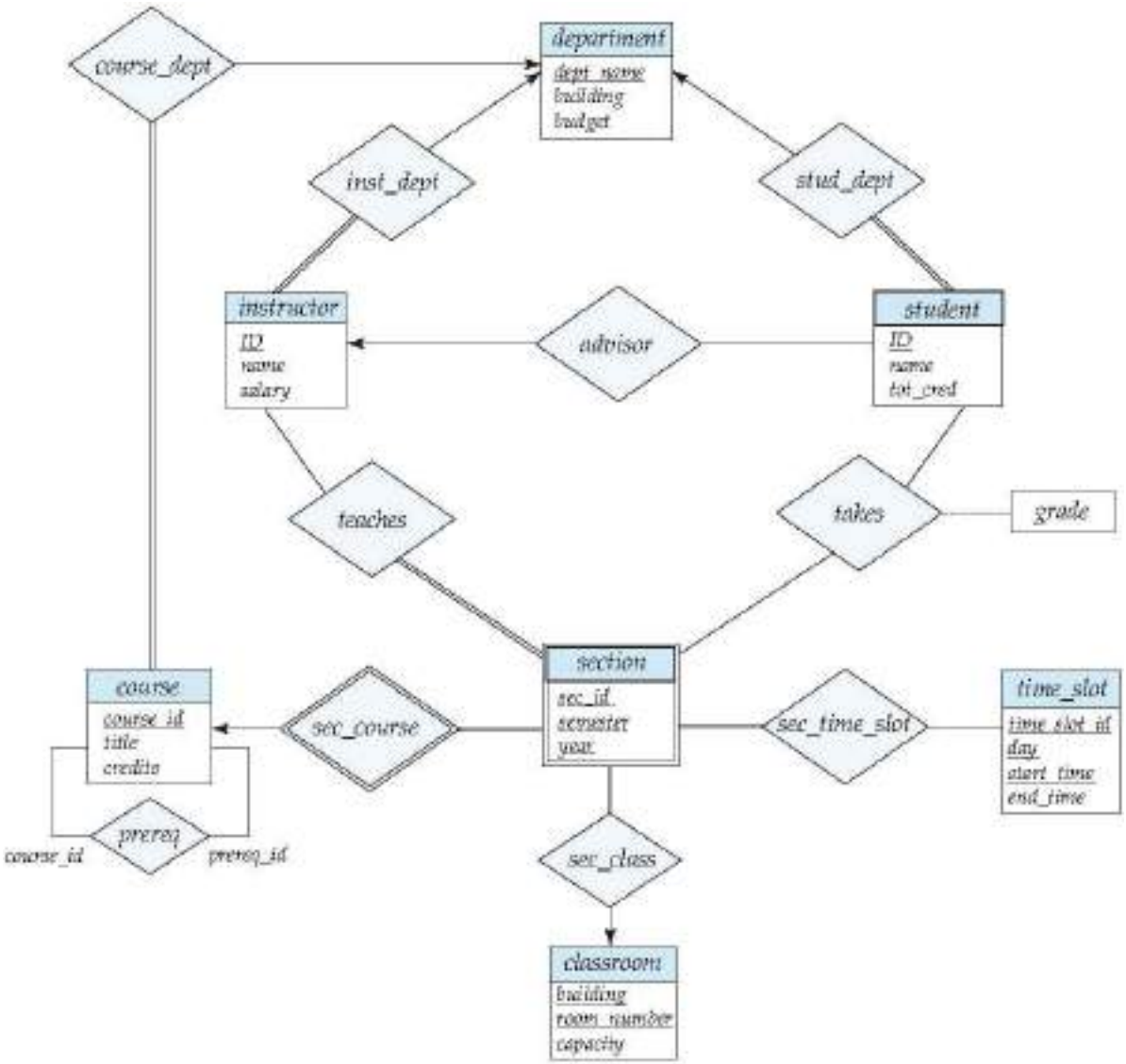
- *ID*
- *name*
  - *first_name*
  - *middle_initial*
  - *last_name*
- *address*
  - *street*
    - *street_number*
    - *street_name*
    - *apt_number*
  - *city*
  - *state*
  - *zip*
- *{ phone_number }*
- *date_of_birth*
- *age ( )*

Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id, sec_id, semester, year*)



E-R Diagram for a University Enterprise



Reduction to Relation Schemas

Reduction to Relation Schemas

- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of schemas.
- For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set.
- Each schema has a number of columns (generally corresponding to attributes), which have unique names.

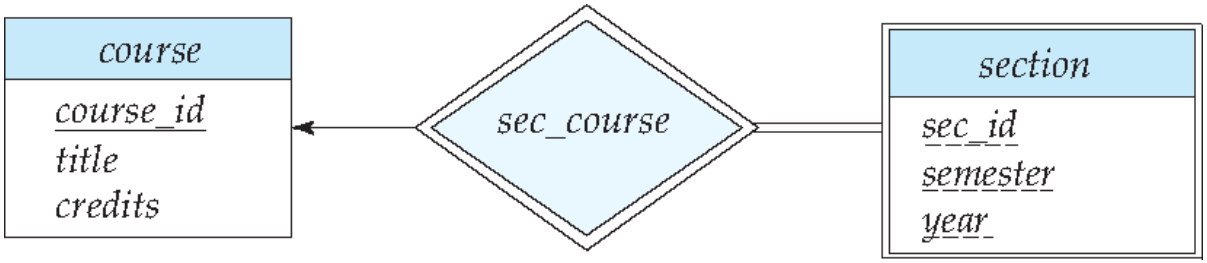
Representing Entity Sets

- A strong entity set reduces to a schema with the same attributes

  student(ID, name, tot_cred)

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

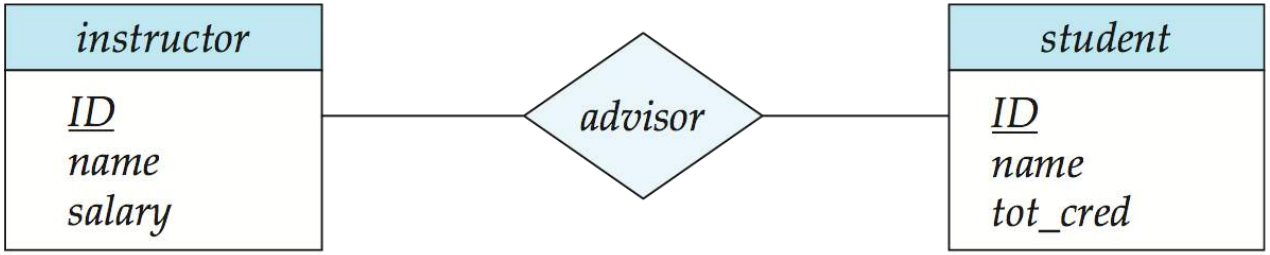
  section (course_id, sec_id, sem, year)



Representing Relationship Sets

- A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- Example: schema for relationship set *advisor*

advisor = (*s_id, i_id*)



Representation of Entity Sets with Composite Attributes

instructor

ID
name
  first_name
  middle_initial
  last_name
address
  street
    street_number
    street_name
    apt_number
  city
  state
  zip
{ phone_number }
date_of_birth
age ( )

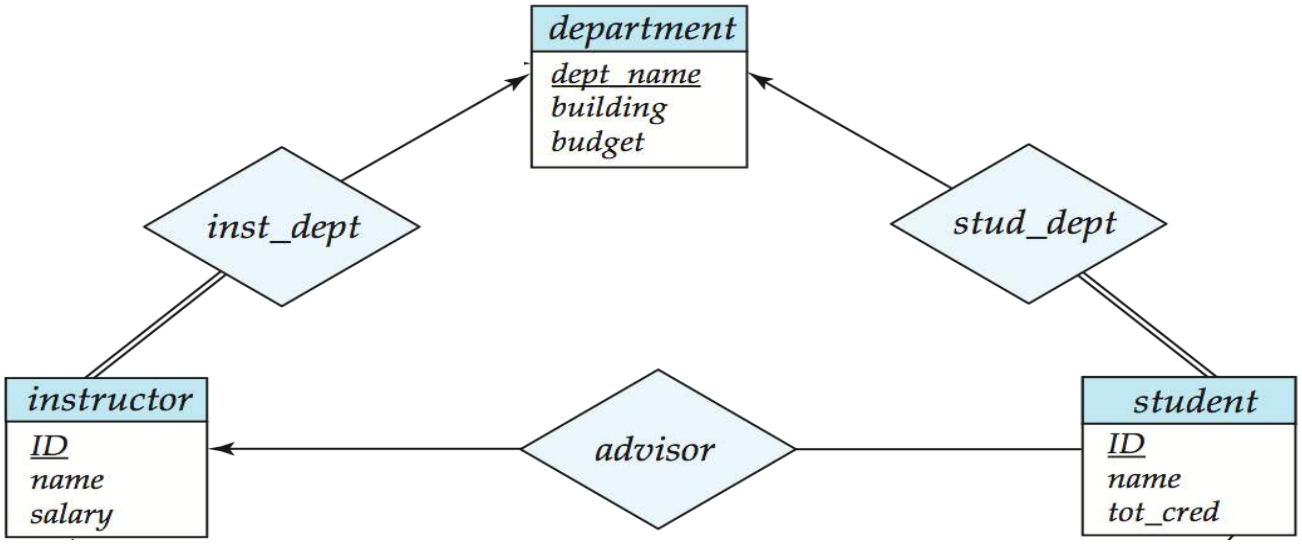
- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - Example: given entity set *instructor* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes *name_first_name* and *name_last_name*
    - Prefix omitted if there is no ambiguity (*name_first_name* could be *first_name)*
- Ignoring multivalued attributes, extended instructor schema is
  - *instructor(ID,*
    first_name, middle_initial, last_name,
    street_number, street_name,
    apt_number, city, state, zip_code,
    date_of_birth)

Representation of Entity Sets with Multivalued Attributes

- A multivalued attribute *M* of an entity *E* is represented by a separate schema *EM*
- Schema *EM* has attributes corresponding to the primary key of *E* and an attribute corresponding to multivalued attribute *M*
- Example: Multivalued attribute *phone_number* of *instructor* is represented by a schema:
  inst_phone= (*ID*, *phone_number*)
- Each value of the multivalued attribute maps to a separate tuple of the relation on schema *EM*
  - For example, an *instructor* entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:
    (22222, 456-7890) and (22222, 123-4567)

Redundancy of Schemas

- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the "many" side, containing the primary key of the "one" side
- Example: Instead of creating a schema for relationship set *inst_dept*, add an attribute *dept_name* to the schema arising from entity set *instructor*



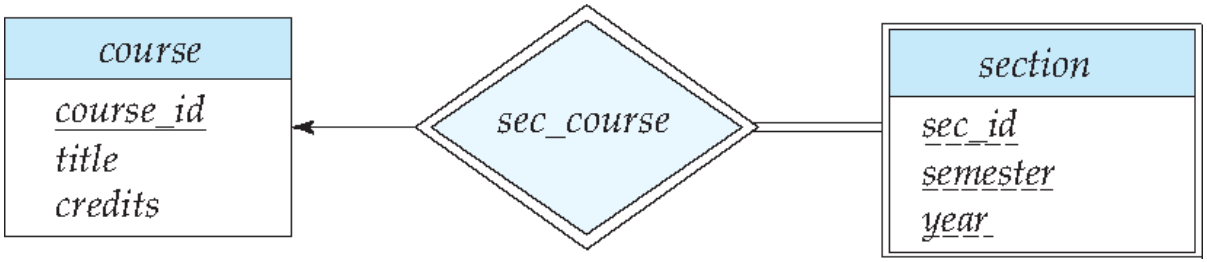
Redundancy of Schemas (Cont.)

- For one-to-one relationship sets, either side can be chosen to act as the “many” side
  - That is, an extra attribute can be added to either of the tables corresponding to the two entity sets
- If participation is *partial* on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values

Redundancy of Schemas (Cont.)

- The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.

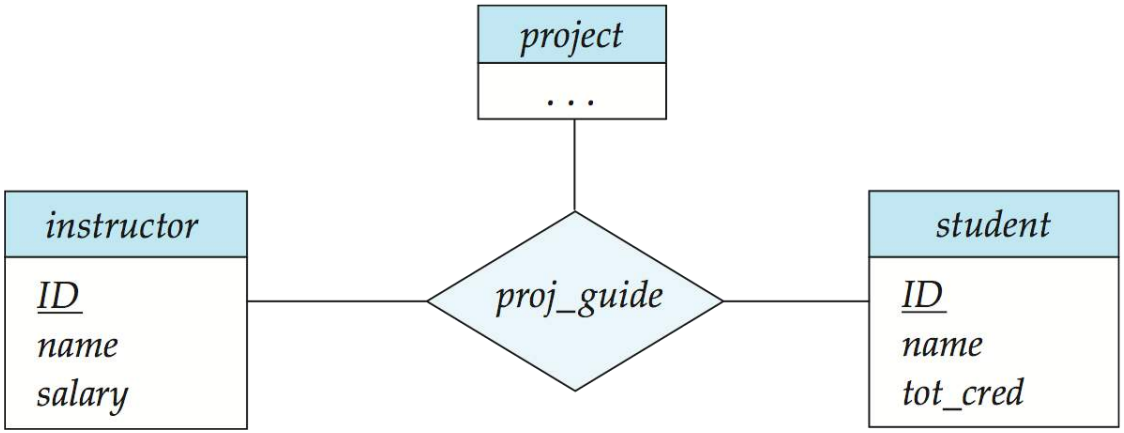
- Example: The *section* schema already contains the attributes that would appear in the *sec_course* schema



Advanced Topics

Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship



Cardinality Constraints on Ternary Relationship

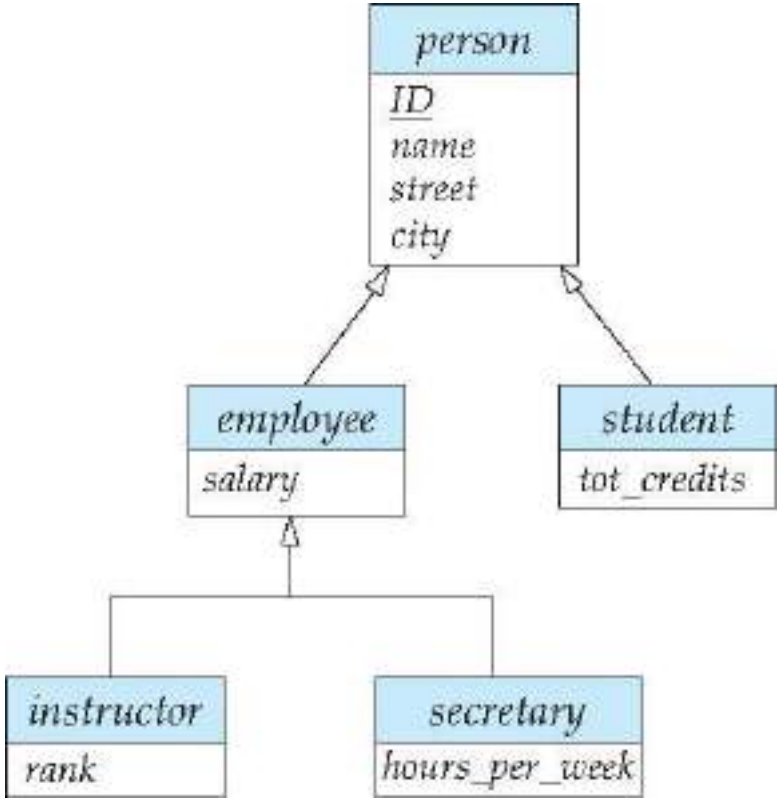
- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- For exampe, an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the meaning.
  - For example, a ternary relationship *R* between *A*, *B* and *C* with arrows to *B* and *C* could mean
    1. Each *A* entity is associated with a unique entity from *B* and *C* or
    2. Each pair of entities from (*A, B*) is associated with a unique *C* entity, and each pair (*A, C*) is associated with a unique *B*
  - Each alternative has been used in different formalisms
  - To avoid confusion we outlaw more than one arrow

Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set.
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* "is a" *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial



Representing Specialization via Schemas

- Method 1:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

  - Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema

Representing Specialization as Schemas (Cont.)

- Method 2:
  - Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

  - Drawback: *name, street* and *city* may be stored redundantly for people who are both students and employees

Generalization

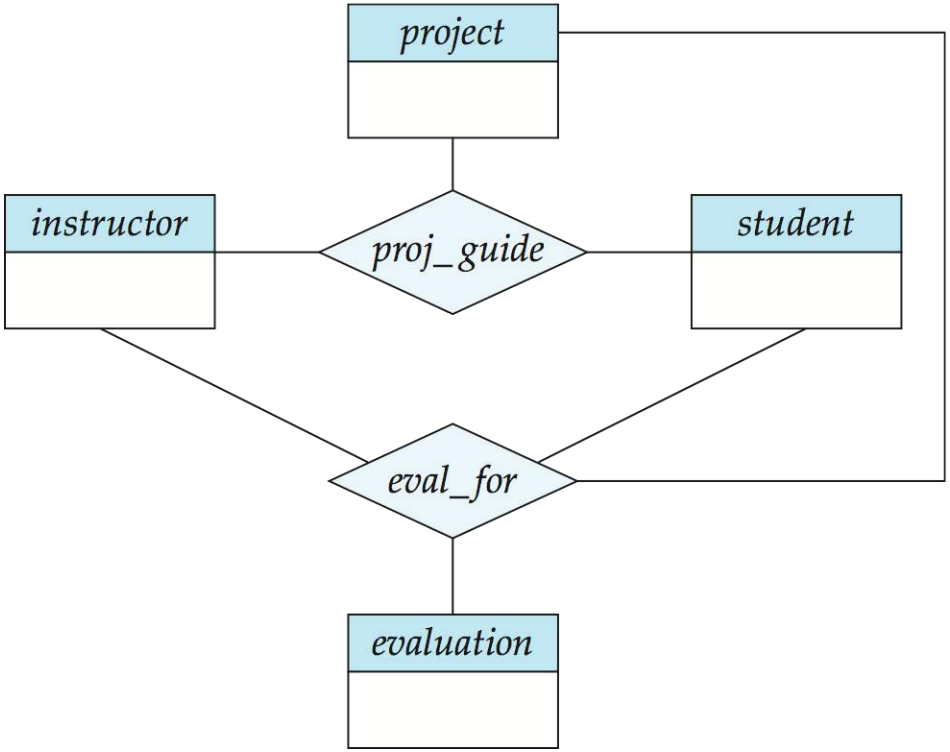
- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

Design Constraints on a Specialization/Generalization

- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong to one of the lower-level entity sets
- Partial generalization is the default. We can specify total generalization in an ER diagram by adding the keyword **total** in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The *student* generalization is total: All student entities must be either graduate or undergraduate. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total

Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project

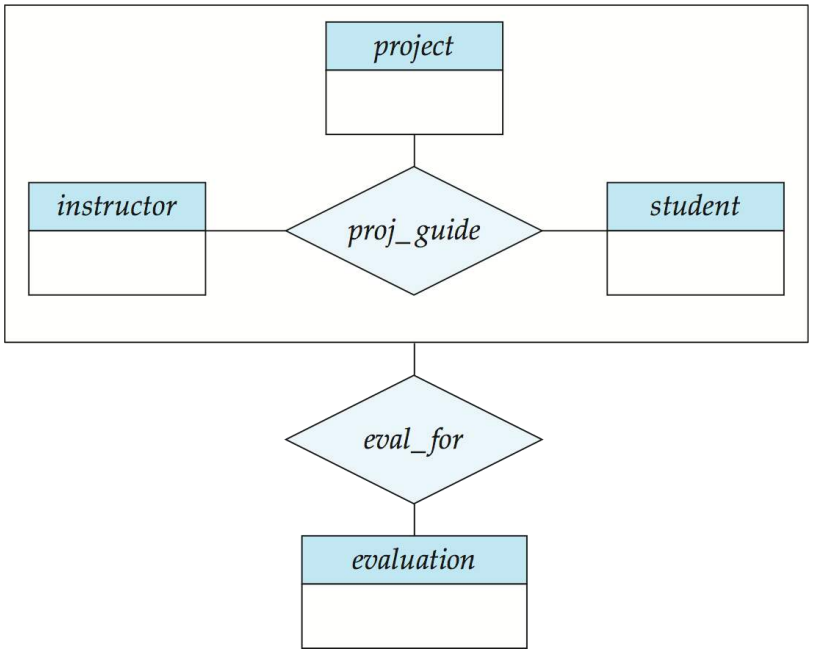


Aggregation (Cont.)

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
  - Every *eval_for* relationship corresponds to a *proj_guide* relationship
  - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
    - So we can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity

Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
  - A student is guided by a particular instructor on a particular project
  - A student, instructor, project combination may have an associated evaluation



Representing Aggregation via Schemas

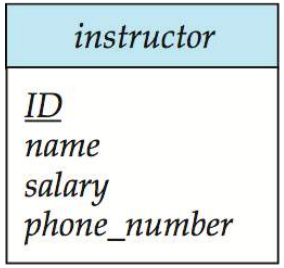
- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval_for* is:

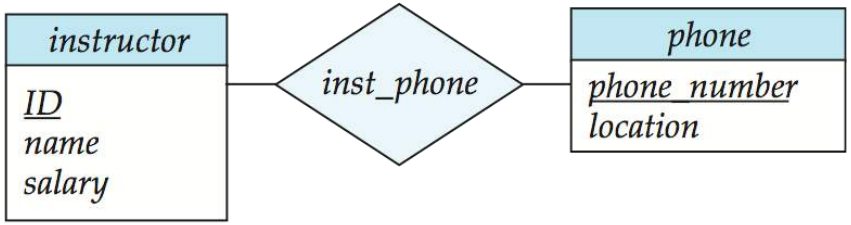
    eval_for (*s_ID, project_id, i_ID, evaluation_id*)
  - The schema *proj_guide* is redundant.

Design Issues

Entities vs. Attributes

- Use of entity sets vs. attributes



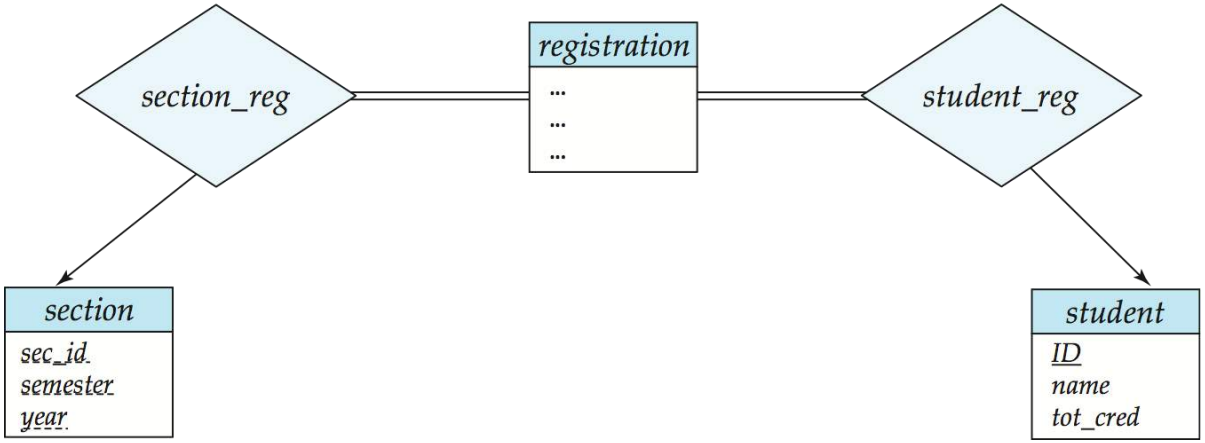


- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)

Entities vs. Relationship sets

- **Use of entity sets vs. relationship sets**

  Possible guideline is to designate a relationship set to describe an action that occurs between entities



- **Placement of relationship attributes**

  For example, attribute date as attribute of advisor or as attribute of student

Binary Vs. Non-Binary Relationships

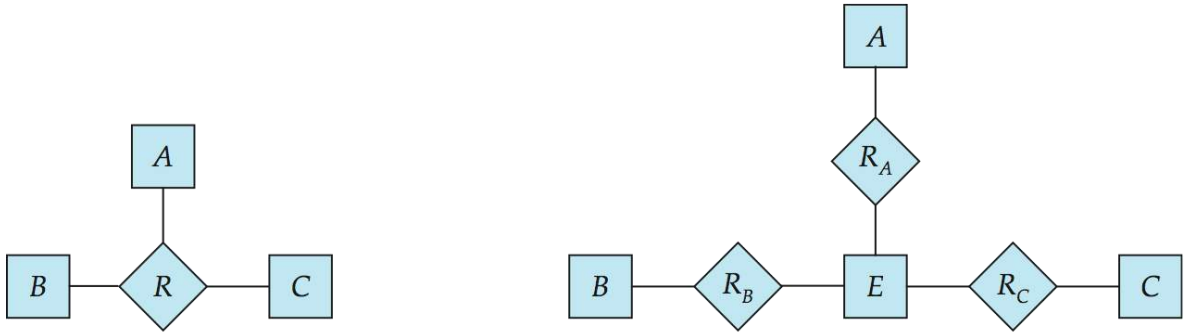
- Although it is possible to replace any non-binary (*n*-ary, for *n* > 2) relationship set by a number of distinct binary relationship sets, a *n*-ary relationship set shows more clearly that several entities participate in a single relationship.
- Some relationships that appear to be non-binary may be better represented using binary relationships
  - For example, a ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
    - Using two binary relationships allows partial information (e.g., only mother being known)
  - But there are some relationships that are naturally non-binary
    - Example: *proj_guide*

Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
  - Replace *R* between entity sets A, B and C by an entity set *E*, and three relationship sets:

    1. R_A, relating *E* and *A*
    2. R_B, relating *E* and *B*
    3. R_C, relating *E* and *C*
  - Create an identifying attribute for *E and* add any attributes of *R* to *E*
  - For each relationship (a_i , b_i , c_i) in *R*, create

    1. a new entity e_i in the entity set *E*
    2. add (e_i , a_i) to R_A
    3. add (e_i , b_i) to R_B
    4. add (e_i , c_i) to R_C



(a) (b)

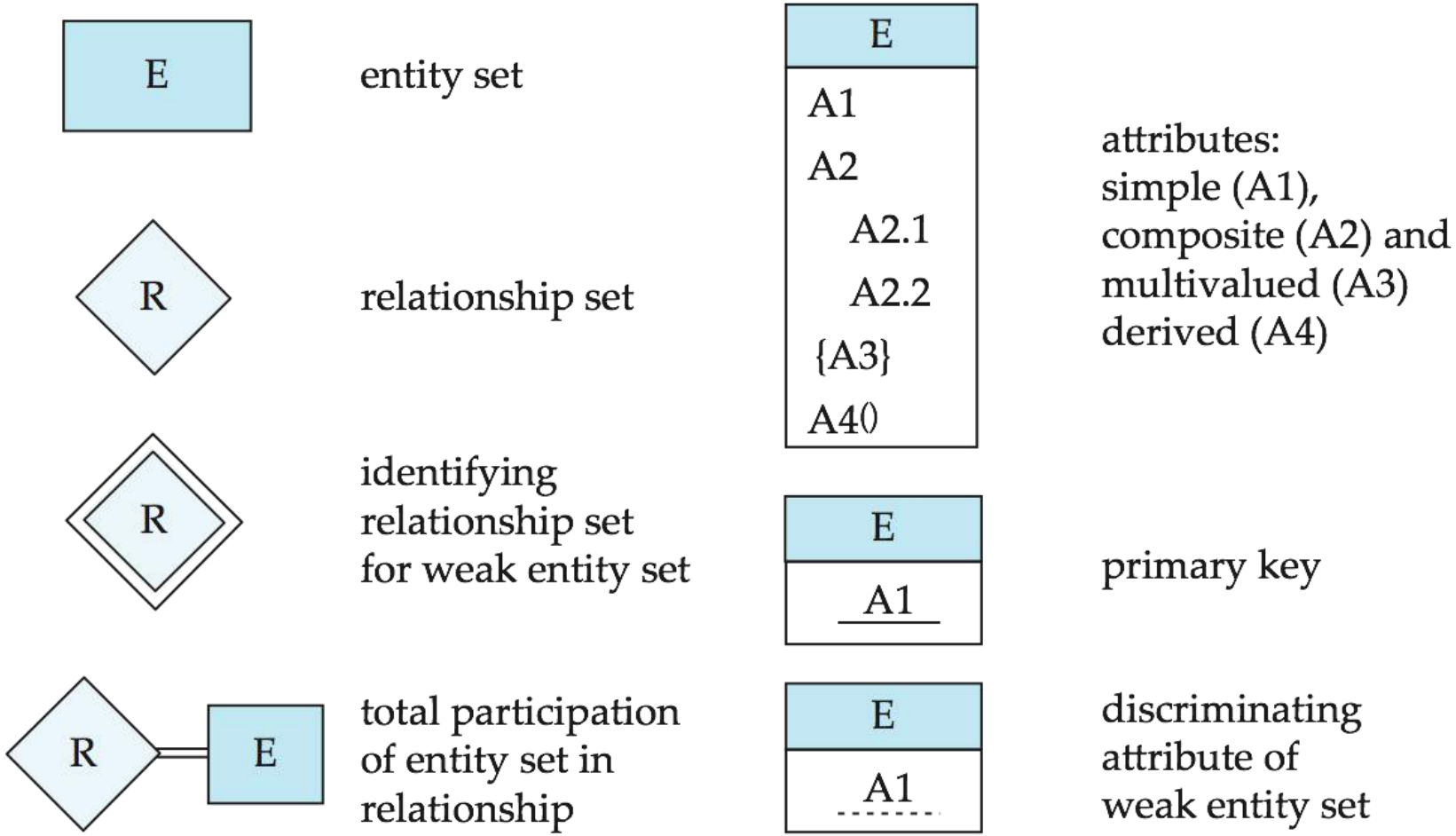
Converting Non-Binary Relationships (Cont.)

- Also need to translate constraints
  - Translating all constraints may not be possible
  - There may be instances in the translated schema that cannot correspond to any instance of *R*
    - Exercise: *add constraints to the relationships* R_A, R_B *and* R_C to ensure that a newly created entity corresponds to exactly one entity in each of entity sets *A, B* and *C*
  - We can avoid creating an identifying attribute by making E a weak entity set (described shortly) identified by the three relationship sets

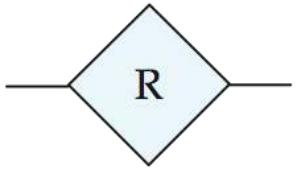
E-R Design Decisions

- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.
- The use of specialization/generalization – contributes to modularity in the design.
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.

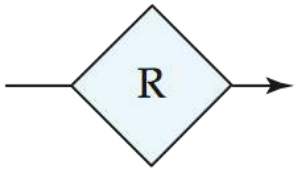
Summary of Symbols Used in E-R Notation



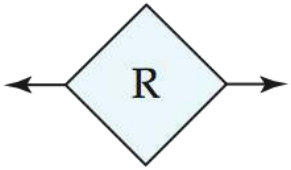
Symbols Used in E-R Notation (Cont.)



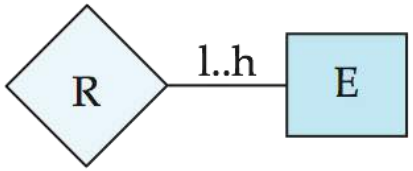
many-to-many relationship



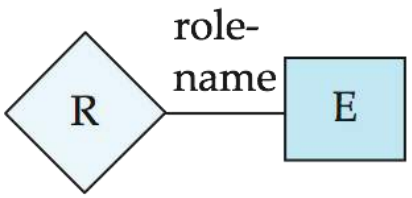
many-to-one relationship



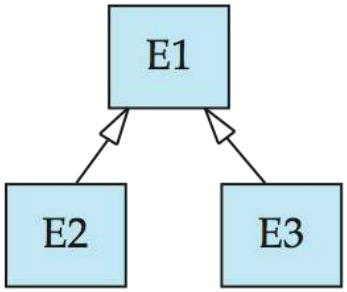
one-to-one relationship



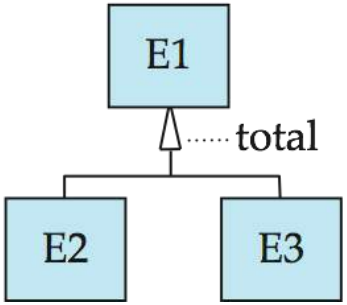
cardinality limits



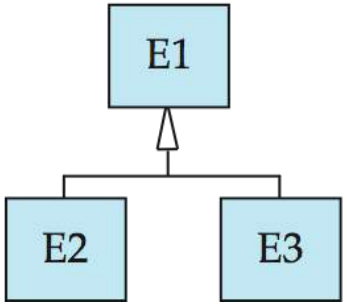
role indicator



ISA: generalization or specialization



total (disjoint) generalization

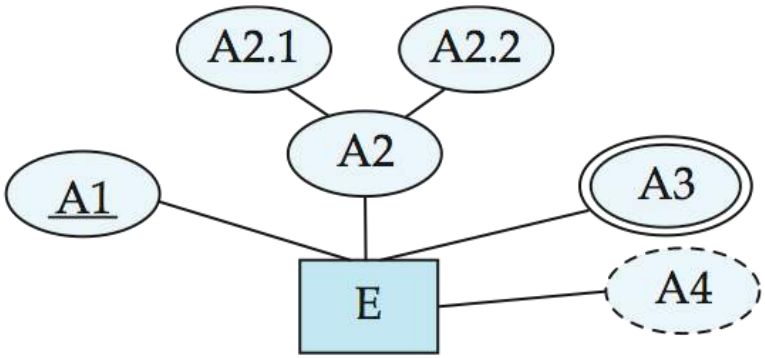


disjoint generalization

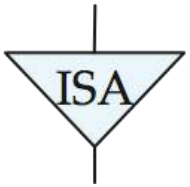
Alternative ER Notations

- Chen, IDE1FX, …

entity set E with
simple attribute A1,
composite attribute A2,
multivalued attribute A3,
derived attribute A4,
and primary key A1



weak entity set

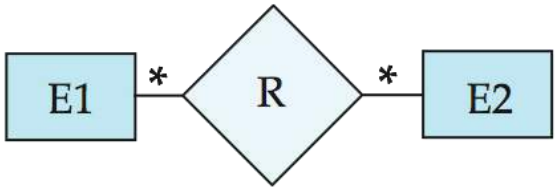
generalization 

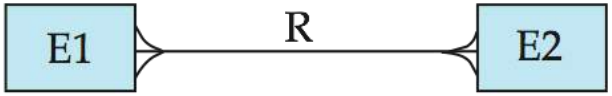
total generalization

Alternative ER Notations

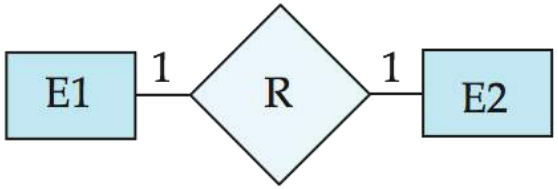
Chen | **IDE1FX (Crows feet notation)**

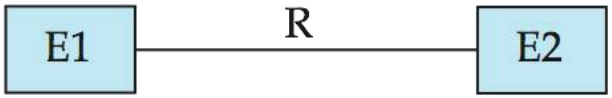
many-to-many relationship



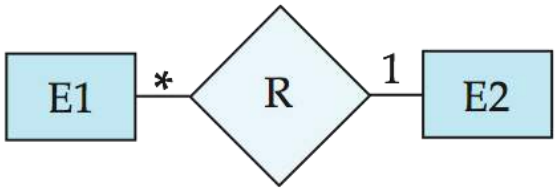


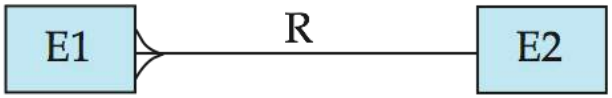
one-to-one relationship



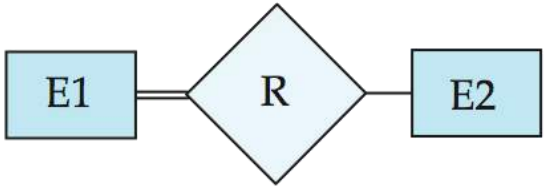


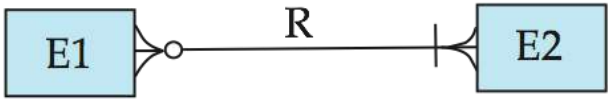
many-to-one relationship





participation in R: total (E1) and partial (E2)



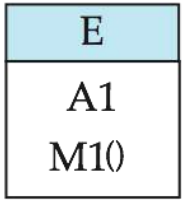


UML

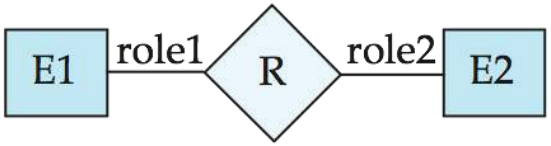
- **UML**: Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.

ER vs. UML Class Diagrams

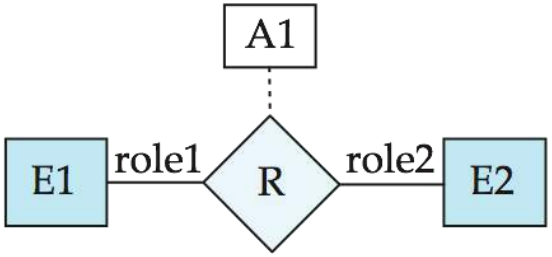
ER Diagram Notation



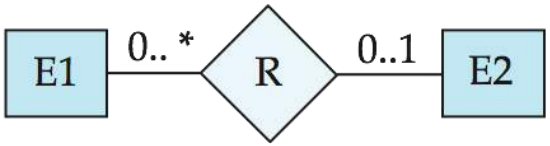
entity with attributes (simple, composite, multivalued, derived)



binary relationship

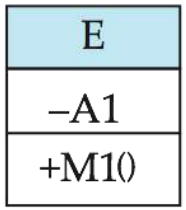


relationship attributes

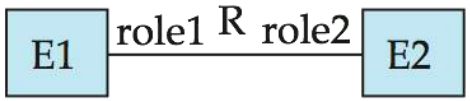


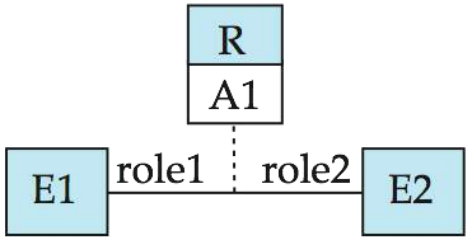
cardinality constraints

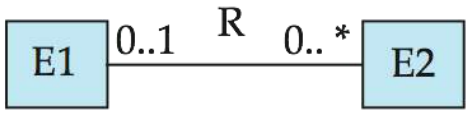
Equivalent in UML



class with simple attributes and methods (attribute prefixes: + = public, – = private, # = protected)



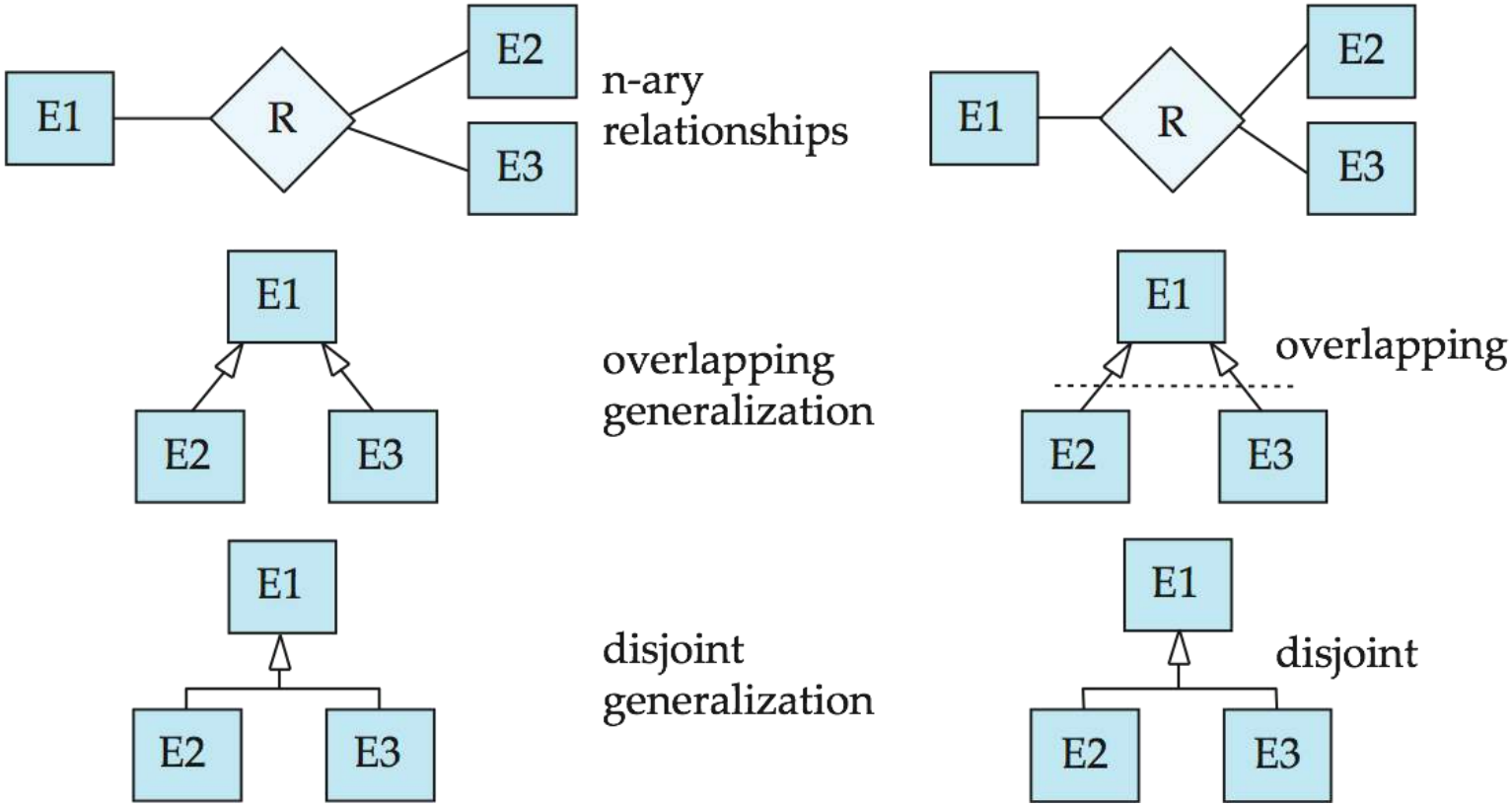




*Note reversal of position in cardinality constraint depiction

ER vs. UML Class Diagrams

ER Diagram Notation | **Equivalent in UML**



*Generalization can use merged or separate arrows independent of disjoint/overlapping

UML Class Diagrams (Cont.)

- Binary relationship sets are represented in UML by just drawing a line connecting the entity sets. The relationship set name is written adjacent to the line.
- The role played by an entity set in a relationship set may also be specified by writing the role name on the line, adjacent to the entity set.
- The relationship set name may alternatively be written in a box, along with attributes of the relationship set, and the box is connected, using a dotted line, to the line depicting the relationship set.

SQL

BBM471 Database Management Systems

SQL Part I: Introduction

Today’s Lecture

1. SQL introduction & schema definitions
2. Basic single-table queries
3. Multi-table queries

1. SQL Introduction & Definitions

What you will learn about in this section

1. What is SQL?
2. Basic schema definitions
3. Keys & constraints intro

SQL Motivation

- Dark times 5 years ago.
  - Are databases dead?
- Now, as before: everyone sells SQL
  - Pig, Hive, Impala
- “Not-Yet-SQL?”

SQL Introduction

- SQL is a standard language for querying and manipulating data
- SQL is a **very high-level** programming language
  - This works because it is optimized well!
- Many standards out there:
  - ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ….
  - Vendors support various subsets

SQL stands for
Structured Query Language

Probably the world's most successful **parallel** programming language (multicore?)

SQL is a…

- Data Definition Language (DDL)
  - Define relational *schemata*
  - Create/alter/delete tables and their attributes

- Data Manipulation Language (DML)
  - Insert/delete/modify tuples in tables
  - Query one or more tables – discussed next!

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A **relation** or **table** is a multiset of tuples having the attributes specified by the schema

Let's break this definition down

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A **multiset** is an unordered list (or: a set with multiple duplicate instances allowed)

List: [1, 1, 2, 3]
Set: {1, 2, 3}
Multiset: {1, 1, 2, 3}

i.e. no *next()*, etc. methods!

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation

*NB: Attributes must have an **atomic** type in standard SQL, i.e. not a list, set, etc.*

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

*Also referred to sometimes as a **record***

A **tuple** or **row** is a single entry in the table having the attributes specified by the schema

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

The number of tuples is the **cardinality** of the relation

The number of attributes is the **arity** of the relation

Data Types in SQL

- Atomic types:
  - Characters: CHAR(20), VARCHAR(50)
  - Numbers: INT, BIGINT, SMALLINT, FLOAT
  - Others: MONEY, DATETIME, …

- Every attribute must have an atomic type
  - Hence tables are flat

Table Schemas

- The **schema** of a table is the table name, its attributes, and their types:

Product(Pname: *string*, Price: *float*, Category: *string*, Manufacturer: *string*)

- A **key** is an attribute whose values are unique; we underline a key

Product(<u>Pname</u>: *string*, Price: *float*, Category: *string*, <u>Manufacturer</u>: *string*)

Key constraints

A <u>key</u> is a **minimal subset of attributes** that acts as a unique identifier for tuples in a relation

- A key is an implicit constraint on which tuples can be in the relation
  - i.e. if two tuples agree on the values of the key, then they must be the same tuple!

Students(sid:string, name:string, gpa: float)

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?

NULL and NOT NULL

- To say "don't know the value" we use NULL
  - NULL has (sometimes painful) semantics, more detail later

Students(sid:string, name:string, gpa: float)

sid	name	gpa
123	Bob	3.9
143	Jim	NULL

Say, Jim just enrolled in his first class.

In SQL, we may constrain a column to be NOT NULL, e.g., "name" in this table

General Constraints

- We can actually specify arbitrary assertions
  - E.g. "*There cannot be 25 people in the DB class*"

- In practice, we don't specify many such constraints. Why?
  - <u>Performance!</u>

Whenever we do something ugly (or avoid doing something convenient) it's for the sake of performance

Summary of Schema Information

- Schema and Constraints are how databases understand the semantics (meaning) of data
- They are also useful for optimization
- SQL supports general constraints:
  - Keys and foreign keys are most important
  - We'll give you a chance to write the others

2. Single-table queries

What you will learn about in this section

1. The SFW query

2. Other useful operators:

    1. LIKE
    2. DISTINCT
    3. ORDER BY

SQL Query

- Basic form (there are many many more bells and whistles)

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

Call this a **SFW** query.

Simple SQL Query: Selection

Selection is the operation of filtering a relation's tuples on some condition

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

Simple SQL Query: Projection

Projection is the operation of producing an output table with tuples that have a subset of their prior attributes

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  Category = 'Gadgets'
```

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks

Notation

Input schema

Product(PName, Price, Category, Manfacturer)

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```

Output schema

Answer(PName, Price, Manfacturer)

A Few Details

- SQL statements are case **insensitive**:
  - Same: SELECT, Select, select
  - Same: Product, product

- **Values** are **not:**
  - <u>Different:</u> ‘Seattle’, ‘seattle’

- Use single quotes for constants:
  - ‘abc’ - yes
  - “abc” - no

LIKE: Simple String Pattern Matching

- *s* ***LIKE*** *p*: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

```
SELECT *
FROM   Products
WHERE  PName LIKE '%gizmo%'
```

DISTINCT: Eliminating Duplicates

```
SELECT DISTINCT Category
FROM   Product
```

Category
Gadgets
Photography
Household

Versus

```
SELECT Category
FROM   Product
```

Category
Gadgets
Gadgets
Photography
Household

ORDER BY: Sorting the Results

```
SELECT   PName, Price, Manufacturer
FROM     Product
WHERE    Category='gizmo' AND Price > 50
ORDER BY Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the **DESC** keyword.

3. Multi-table queries

What you will learn about in this section

1. Foreign key constraints
2. Joins: basics
3. Joins: SQL semantics

Foreign Key constraints

- Suppose we have the following schema:

Students(sid: *string*, name: *string*, gpa: *float*)

Enrolled(student_id: *string*, cid: *string*, grade: *string*)

- And we want to impose the following constraint:
  - 'Only bona fide students may enroll in courses' i.e. a student must appear in the Students table to enroll in a class

student_id alone is not a key- what is?

Students

sid	name	gpa
101	Bob	3.2
123	Mary	3.8

Enrolled

student_id	cid	grade
123	564	A
123	537	A+

We say that student_id is a **foreign key** that refers to Students

Declaring Foreign Keys

Students(sid: *string,* name: *string,* gpa: *float*)
Enrolled(student_id: *string*, cid: *string,* grade: *string*)

```
CREATE TABLE Enrolled(
        student_id CHAR(20),
        cid                CHAR(20),
        grade    CHAR(10),
        PRIMARY KEY (student_id, cid),
        FOREIGN KEY (student_id) REFERENCES Students(sid)
)
```

Foreign Keys and update operations

Students(sid: *string*, name: *string*, gpa: *float*)

Enrolled(student_id: *string*, cid: *string*, grade: *string*)

- What if we insert a tuple into Enrolled, but no corresponding student?
  - INSERT is rejected (foreign keys are constraints)!

- What if we delete a student?

DBA chooses (syntax in the book)

  1. Disallow the delete
  2. Remove all of the courses for that student
  3. *SQL allows a third via NULL (not yet covered)*

Keys and Foreign Keys

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is a foreign key vs. a key here?

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

Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
           AND Country='Japan'
       AND Price <= 200
```

Note: we will often omit attribute types in schema definitions for brevity, but assume attributes are always atomic types

Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
   AND Price <= 200
```

A **join** between tables returns all unique combinations of their tuples **which meet some specified join condition**

Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Several equivalent ways to write a basic join in SQL:

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
           AND Country='Japan'
    AND Price <= 200
```

```
SELECT PName, Price
FROM   Product
JOIN   Company ON Manufacturer = Cname
                  AND Country='Japan'
WHERE  Price <= 200
```

A few more later on…

Joins

Product

PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
       AND Price <= 200
```

PName	Price
SingleTouch	$149.99

Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)

Company(name, address)

```
SELECT DISTINCT name, address
FROM            Person, Company
WHERE           worksfor = name
```

Which "address" does this refer to?

Which "name"s??

Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)

Company(name, address)

Both equivalent ways to resolve variable ambiguity

```
SELECT DISTINCT Person.name, Person.address
FROM            Person, Company
WHERE           Person.worksfor = Company.name
```

```
SELECT DISTINCT p.name, p.address
FROM            Person p, Company c
WHERE           p.worksfor = c.name
```

Meaning (Semantics) of SQL Queries

```
SELECT x1.a1, x1.a2, …, xn.ak
FROM   R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions(x1,…, xn)
```

Almost never the *fastest* way to compute it!

```
Answer = {}
for x1 in R1 do
    for x2 in R2 do
        …..
            for xn in Rn do
                if Conditions(x1,…, xn)
                    then Answer = Answer ∪ {(x1.a1, x1.a2, …, xn.ak)}
return Answer
```

Note: this is a *multiset* union

An example of SQL semantics

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

A
1
3

B	C
2	3
3	4
3	5

Cross Product

A	B	C
1	2	3
1	3	4
1	3	5
3	2	3
3	3	4
3	3	5

Apply Selections / Conditions

A	B	C
3	3	4
3	3	5

Apply Projection

Output

A
3
3

Note the *semantics* of a join

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

1. Take **cross product**:

$$X = R \times S$$

Recall: Cross product (A X B) is the set of all unique tuples in A,B

Ex: {a,b,c} X {1,2}
= {(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)}

2. Apply **selections / conditions**:

$$Y = \{(r, s) \in X \mid r.A == r.B\}$$

= Filtering!

3. Apply **projections** to get final output:

$$Z = (y.A,)\ for\ y \in Y$$

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on…)

Note: we say “semantics” not “execution order”

- The preceding slides show *what a join means*
- Not actually how the DBMS executes it under the cover

A Subtlety about Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Find all countries that manufacture some product in the 'Gadgets' category.

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=CName AND Category='Gadgets'
```

A subtlety about Joins

Product

PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=Cname
  AND Category='Gadgets'
```

Country
?
?

What is the problem ?
What's the solution ?

An Unintuitive Query

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

What does it compute?

SQL Part II: Advanced

Today’s Lecture

1. Set operators & nested queries
2. Aggregation & GROUP BY
3. Advanced SQL-izing

1. Set Operators & Nested Queries

What you will learn about in this section

1. Multiset operators in SQL
2. Nested queries

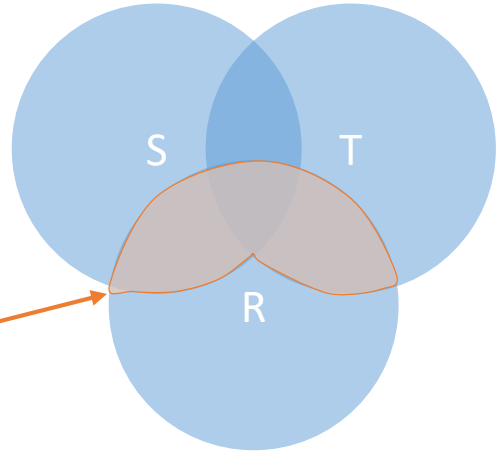
An Unintuitive Query

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

What does it compute?

An Unintuitive Query

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```



Computes $R \cap (S \cup T)$

But what if $S = \phi$?

Go back to the semantics!

An Unintuitive Query

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

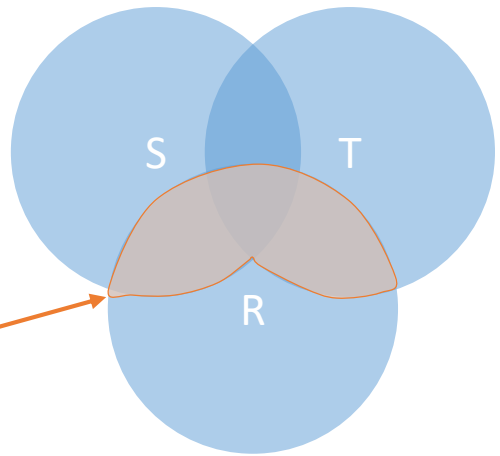
- Recall the semantics!
  1. Take cross-product
  2. Apply selections / conditions
  3. Apply projection
- If S = {}, then the cross product of R, S, T = {}, and the query result = {}!

Must consider semantics here.
Are there more explicit way to do set operations like this?

What does this look like in Python?

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

$R \cap (S \cup T)$



- Semantics:
  1. Take <u>cross-product</u>
  2. Apply <u>selections</u> / <u>conditions</u>
  3. Apply <u>projection</u>

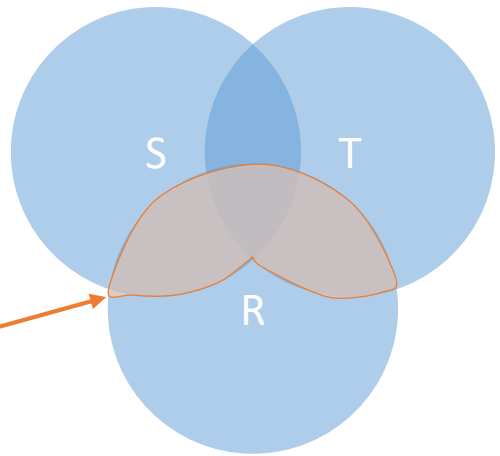
Joins / cross-products are just **nested for loops** (in simplest implementation)!

If-then statements!

What does this look like in Python?

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

$R \cap (S \cup T)$




```
output = {}

for r in R:
  for s in S:
    for t in T:
      if r['A'] == s['A'] or r['A'] == t['A']:
        output.add(r['A'])
return list(output)
```

Can you see now what happens if S = []?

Multiset Operations

Recall Multisets

Multiset X

Tuple
(1, a)
(1, a)
(1, b)
(2, c)
(2, c)
(2, c)
(1, d)
(1, d)

Equivalent Representations of a **Multiset**

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	1
(2, c)	3
(1, d)	2

$\boldsymbol{\lambda(X)}$= *"Count of tuple in X" (Items not listed have implicit count 0)*

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cap$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	2
(1, b)	0
(2, c)	2
(1, d)	0

$$\boldsymbol{\lambda(Z) = min(\lambda(X), \lambda(Y))}$$

For sets, this is **intersection**

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cup$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

$$\lambda(Z) = \lambda(X) + \lambda(Y)$$

For sets,
this is **union**

Multiset Operations in SQL

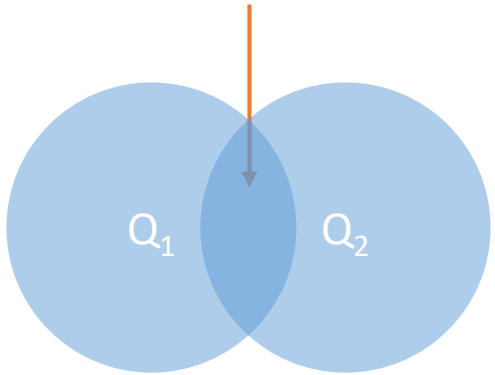
Explicit Set Operators: INTERSECT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A

INTERSECT

SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \cap \{r.A \mid r.A = t.A\}$$



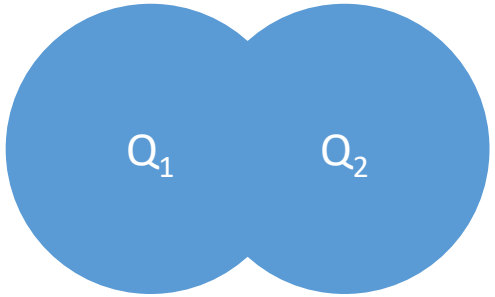
UNION

```
SELECT  R.A
FROM   R, S
WHERE  R.A=S.A

UNION

SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$$



Why aren't there duplicates?

What if we want duplicates?

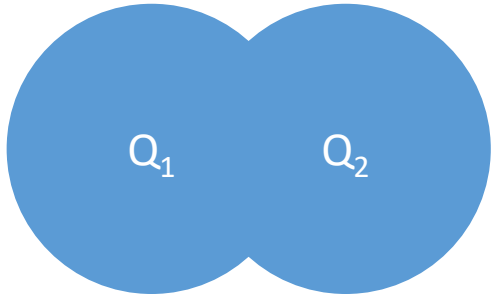
UNION ALL

```
SELECT  R.A
FROM   R, S
WHERE  R.A=S.A

UNION ALL

SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$$



ALL indicates Multiset operations

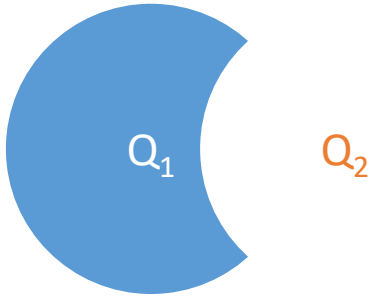
EXCEPT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A

EXCEPT

SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \backslash \{r.A | r.A = t.A\}$$



What is the multiset version?

INTERSECT: Still some subtle problems…

Company(name, hq_city)
Product(pname, maker, factory_loc)

```
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
       AND factory_loc = 'US'

INTERSECT

SELECT hq_city
FROM   Company, Product
WHERE  maker = name
                AND factory_loc = 'China'
```

*"Headquarters of companies which make products in US **AND** China"*

What if two companies have HQ in US: BUT one has factory in China (but not US) and vice versa? **What goes wrong?**

INTERSECT: Remember the semantics!

Company(name, hq_city) AS C
Product(pname, maker, factory_loc) AS P

```
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
  AND factory_loc='US'
INTERSECT
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
AND factory_loc='China'
```

Example: C JOIN P on maker = name

C.name	C.hq_city	P.pname	P.maker	P.factory_loc
X Co.	Seattle	X	X Co.	U.S.
Y Inc.	Seattle	X	Y Inc.	China

INTERSECT: Remember the semantics!

Company(name, hq_city) AS C
Product(pname, maker, factory_loc) AS P

```
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
  AND factory_loc='US'
INTERSECT
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
AND factory_loc='China'
```

Example: C JOIN P on maker = name

C.name	C.hq_city	P.pname	P.maker	P.factory_loc
X Co.	Seattle	X	X Co.	U.S.
Y Inc.	Seattle	X	Y Inc.	China

X Co has a factory in the US (but not China)
Y Inc. has a factory in China (but not US)

But Seattle is returned by the query!

We did the INTERSECT on the wrong attributes!

One Solution: **Nested Queries**

Company(name, hq_city)
Product(pname, maker, factory_loc)

```
SELECT DISTINCT hq_city
FROM   Company, Product
WHERE  maker = name
     AND name IN (
                    SELECT maker
                    FROM   Product
                    WHERE  factory_loc = 'US')
          AND name IN (
                    SELECT maker
                    FROM   Product
                    WHERE  factory_loc = 'China')
```

*"Headquarters of companies which make products in US **AND** China"*

Note: If we hadn't used DISTINCT here, how many copies of each hq_city would have been returned?

High-level note on nested queries

- We can do nested queries because SQL is ***compositional:***
  - Everything (inputs / outputs) is represented as multisets- the output of one query can thus be used as the input to another (nesting)!
- This is <u>extremely</u> powerful!

Nested queries: Sub-queries Returning Relations

Another example:

Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)

```
SELECT c.city
FROM   Company c
WHERE  c.name  IN (
          SELECT pr.maker
     FROM   Purchase p, Product pr
     WHERE  p.product = pr.name
            AND p.buyer = 'Joe Blow')
```

"Cities where one can find companies that manufacture products bought by Joe Blow"

Nested Queries

Is this query equivalent?

```
SELECT c.city
FROM   Company c,
       Product pr,
       Purchase p
WHERE  c.name = pr.maker
  AND  pr.name = p.product
  AND  p.buyer = 'Joe Blow'
```

Beware of duplicates!

Nested Queries

```
SELECT DISTINCT c.city
FROM   Company c,
       Product pr,
       Purchase p
WHERE  c.name = pr.maker
  AND  pr.name = p.product
  AND  p.buyer = 'Joe Blow'
```

```
SELECT DISTINCT c.city
FROM   Company c
WHERE  c.name  IN (
  SELECT pr.maker
  FROM   Purchase p, Product pr
  WHERE  p.product = pr.name
         AND p.buyer = 'Joe Blow')
```

Now they are equivalent

Subqueries Returning Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

ANY and ALL not supported by SQLite.

Ex: Product(name, price, category, maker)

```
SELECT name
FROM   Product
WHERE  price > ALL(
          SELECT price
   FROM   Product
   WHERE  maker = 'Gizmo-Works')
```

Find products that are more expensive than all those produced by "Gizmo-Works"

Subqueries Returning Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- <u>EXISTS R</u>

Ex: Product(name, price, category, maker)

```
SELECT p1.name
FROM   Product p1
WHERE  p1.maker = 'Gizmo-Works'
  AND EXISTS(
            SELECT p2.name
      FROM   Product p2
      WHERE  p2.maker <> 'Gizmo-Works'
               AND p1.name = p2.name)
```

<> means !=

Find 'copycat' products, i.e. products made by competitors with the same names as products made by "Gizmo-Works"

Nested queries as alternatives to INTERSECT and EXCEPT

INTERSECT and EXCEPT not in some DBMSs!

```
(SELECT R.A, R.B
 FROM   R)
INTERSECT
(SELECT S.A, S.B
 FROM   S)
```

⇨

```
SELECT R.A, R.B
FROM   R
WHERE EXISTS(
            SELECT *
            FROM S
      WHERE R.A=S.A AND R.B=S.B)
```

If R, S have no duplicates, then can write without sub-queries (HOW?)

```
(SELECT R.A, R.B
 FROM   R)
EXCEPT
(SELECT S.A, S.B
 FROM   S)
```

⇨

```
SELECT R.A, R.B
FROM   R
WHERE NOT EXISTS(
             SELECT *
      FROM S
      WHERE R.A=S.A AND R.B=S.B)
```

A question for Database Fans & Friends

- Can we express the previous nested queries as single SFW queries?

- Hint: show that all SFW queries are monotone (roughly: more tuples, more answers).
  - A query with **ALL** is often not monotone

Correlated Queries

Movie(title, year, director, length)

```
SELECT DISTINCT title
FROM   Movie AS m
WHERE  year <> ANY(
                SELECT  year
        FROM    Movie
        WHERE  title =  m.title)
```

Find movies whose title appears more than once.

Complex Correlated Query

Product(name, price, category, maker, year)

```
SELECT DISTINCT  x.name, x.maker
FROM   Product AS x
WHERE  x.price > ALL(
                    SELECT y.price
        FROM   Product AS y
        WHERE  x.maker = y.maker
                     AND y.year < 1972)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Can be very powerful (also much harder to optimize)

Basic SQL Summary

- SQL provides a high-level declarative language for manipulating data (DML)
- The workhorse is the SFW block
- Set operators are powerful but have some subtleties
- Powerful, nested queries also allowed.

2. Aggregation & GROUP BY

What you will learn about in this section

1. Aggregation operators
2. GROUP BY
3. GROUP BY: with HAVING, semantics

Aggregation

```
SELECT AVG(price)
FROM   Product
WHERE  maker = "Toyota"
```

```
SELECT COUNT(*)
FROM   Product
WHERE  year > 1995
```

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

Except COUNT, all aggregations apply to a single attribute

Aggregation: COUNT

- COUNT applies to duplicates, unless otherwise stated

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

Note: Same as COUNT(*). Why?

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
```

What do these mean?

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

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 (= 1*20 + 1.50*20)

Grouping and Aggregation

Purchase(product, date, price, quantity)

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

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

Let's see what this means…

Grouping and Aggregation

Semantics of the query:

1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

1. Compute the FROM and WHERE clauses

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

FROM

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

2. Group by the attributes in the GROUP BY

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

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

GROUP BY

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

3. Compute the SELECT clause: grouped attributes and aggregates

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

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

SELECT

Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
HAVING   SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples…***

General form of Grouping and Aggregation

```
SELECT     S
FROM       R1,…,Rn
WHERE      C1
GROUP BY   a1,…,ak
HAVING     C2
```

- S = Can **ONLY** contain attributes $a_1,\ldots,a_k$ and/or aggregates over other attributes
- C_1 = is any condition on the attributes in $R_1,\ldots,R_n$
- C_2 = is any condition on the aggregate expressions

General form of Grouping and Aggregation

```
SELECT     S
FROM       R1,…,Rn
WHERE      C1
GROUP BY   a1,…,ak
HAVING     C2
```

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply condition C_2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Group-by v.s. Nested Query

Author(<u>login</u>, name)

Wrote(login, url)

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM   Author
WHERE  COUNT(
          SELECT Wrote.url
   FROM   Wrote
   WHERE  Author.login = Wrote.login) > 10
```

This is SQL by a novice

Group-by v.s. Nested Query

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT   Author.name
FROM     Author, Wrote
WHERE    Author.login = Wrote.login
GROUP BY Author.name
HAVING   COUNT(Wrote.url) > 10
```

This is SQL by an expert

No need for DISTINCT: automatically from GROUP BY

Group-by vs. Nested Query

Which way is more efficient?

- Attempt #1- *With nested:* How many times do we do a SFW query over all of the Wrote relations?

- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be **<u>much</u>** more efficient!

3. Advanced SQL-izing

What you will learn about in this section

1. Quantifiers
2. NULLs
3. Outer Joins

Quantifiers

Product(name, price, company)
Company(name, city)

```
SELECT DISTINCT Company.cname
FROM   Company, Product
WHERE  Company.name = Product.company
  AND Product.price < 100
```

Find all companies that make <u>some</u> products with price < 100

An **<u>existential quantifier</u>** is a logical quantifier (roughly) of the form “there exists”

Quantifiers

Product(name, price, company)
Company(name, city)

```
SELECT DISTINCT Company.cname
FROM   Company
WHERE  Company.name NOT IN(
          SELECT Product.company
          FROM Product.price >= 100)
```

A <u>universal quantifier</u> is of the form “for all”

Find all companies with products <u>all</u> having price < 100

Equivalent

Find all companies that make <u>only</u> products with price < 100

NULLS in SQL

- Whenever we don’t have a value, we can put a NULL

- Can mean many things:
  - Value does not exists
  - Value exists but is unknown
  - Value not applicable
  - Etc.

- The schema specifies for each attribute if can be null (*nullable* attribute) or not

- How does SQL cope with tables that have NULLs?

Null Values

- *For numerical operations,* NULL -> NULL:
  - If x = NULL then 4*(3-x)/7 is still NULL

- *For boolean operations,* in SQL there are three values:

FALSE	**=**	**0**
TRUE	**=**	**1**
UNKNOWN		

  - If x = NULL then x =“Joe” is UNKNOWN

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM   Person
WHERE (age < 25)
  AND (height > 6 AND weight > 190)
```

Won't return e.g. (age=20 height=NULL weight=200)!

Rule in SQL: include only tuples that yield TRUE (1.0)

Null Values

Unexpected behavior:

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
```

Some Persons are not included !

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
   OR age IS NULL
```

Now it includes all Persons!

RECAP: Inner Joins

By default, joins in SQL are **“inner joins”:**

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
  JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

INNER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
  INNER JOIN Purchase
            ON Product.name = Purchase.prodName
```

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Products that never sold (with no Purchase tuple) will be lost!

Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
  - I.e. If we join relations A and B on a.X = b.X, and there is an entry in A with X=5, but none in B with X=5…
    - A LEFT OUTER JOIN will return a tuple (a, NULL)!

- Left outer joins in SQL:

```
SELECT Product.name, Purchase.store
FROM   Product
 LEFT OUTER JOIN Purchase ON
            Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

LEFT OUTER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase
            ON Product.name = Purchase.prodName
```

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Other Outer Joins

- Left outer join:
  - Include the left tuple even if there’s no match
- Right outer join:
  - Include the right tuple even if there’s no match
- Full outer join:
  - Include the both left and right tuples even if there’s no match

Summary

SQL is a rich programming language that handles the way data is processed *declaratively*

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the *CS145 Introduction to Databases* lecture given by *Christopher Ré* at *Stanford University* (http://web.stanford.edu/class/cs145/).

Relational Algebra

Today's Lecture

1. The Relational Model & Relational Algebra

2. Relational Algebra Pt. II

1. The Relational Model & Relational Algebra

What you will learn about in this section

1. The Relational Model
2. Relational Algebra: Basic Operators
3. Execution

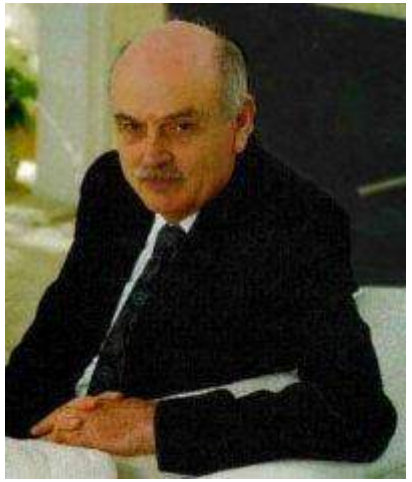
Motivation

The Relational model is **precise**, **implementable**, and we can operate on it (query/update, etc.)

Database maps internally into this *procedural language*.

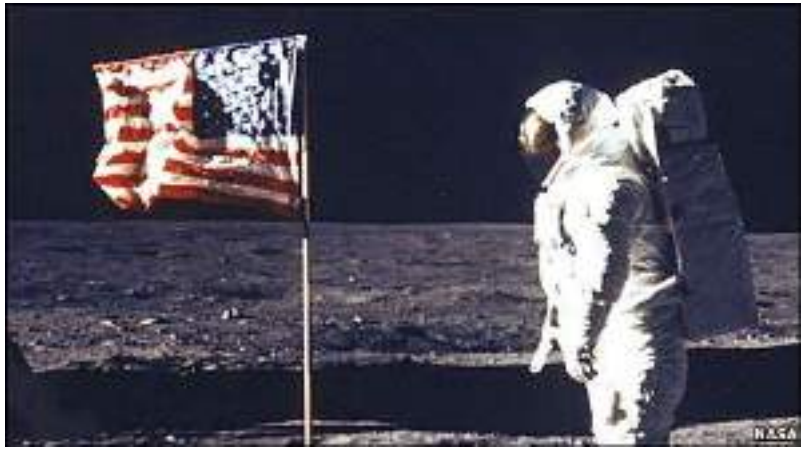
A Little History

- Relational model due to Edgar "Ted" Codd, a mathematician at IBM in 1970
  - A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM* **13** (6): 377–387



Won Turing award 1981

- IBM didn't want to use relational model
  - *Apparently used in the moon landing…*
  - *Google for "***IMS and the Apollo program***"*



The Relational Model: Schemata

- Relational Schema:

Students(sid: *string*, name: *string*, gpa: *float*)

Relation name

String, float, int, etc. are the **domains** of the attributes

Attributes

The Relational Model: Data

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation

Student

sid	name	gpa
001	Bob	3.2
002	Joe	2.8
003	Mary	3.8
004	Alice	3.5

The number of attributes is the **arity** of the relation

The Relational Model: Data

Student

sid	name	gpa
001	Bob	3.2
002	Joe	2.8
003	Mary	3.8
004	Alice	3.5

The number of tuples is the **cardinality** of the relation

A **tuple** or **row** (or *record)* is a single entry in the table having the attributes specified by the schema

The Relational Model: Data

Student

sid	name	gpa
001	Bob	3.2
002	Joe	2.8
003	Mary	3.8
004	Alice	3.5

Recall: In practice DBMSs relax the set requirement, and use multisets.

A **relational instance** is a ***set*** of tuples all conforming to the same *schema*

To Reiterate

- A *relational schema* describes the data that is contained in a *relational instance*

Let $R(f_1{:}Dom_1,\ldots,f_m{:}Dom_m)$ be a *relational schema* then, an *instance* of R is a subset of $Dom_1 \times Dom_2 \times \ldots \times Dom_n$

In this way, a *relational schema* R is a **total function from attribute *names* to types**

One More Time

- A *relational schema* describes the data that is contained in a *relational instance*

A relation R of arity *t* is a function:
$R : Dom_1 \times \ldots \times Dom_t \rightarrow \{0,1\}$

i.e. returns whether or not a tuple of matching types is a member of it

Then, the schema is simply the *signature* of the function

Note here that order matters, attribute name doesn't…
We'll (mostly) work with the other model (last slide) in which **attribute name matters, order doesn't!**

A relational database

- A *relational database schema* is a set of relational schemata, one for each relation

- A *relational database instance* is a set of relational instances, one for each relation

Two conventions:

1. We call relational database instances as simply ***databases***
2. We assume all instances are valid, i.e., satisfy the *domain constraints*

Remember the CMS

- *Relation DB Schema*
  - Students(sid: *string*, name: *string*, gpa: *float*)
  - Courses(cid: *string*, cname: *string*, credits: *int*)
  - Enrolled(sid: *string,* cid*: string,* grade*: string*)

Note that the schemas impose effective <u>domain / type constraints</u>, i.e. Gpa can't be "Apple"

Relation Instances

Sid	Name	Gpa
101	Bob	3.2
123	Mary	3.8

Students

cid	cname	credits
564	564-2	4
308	417	2

Courses

sid	cid	Grade
123	564	A

Enrolled

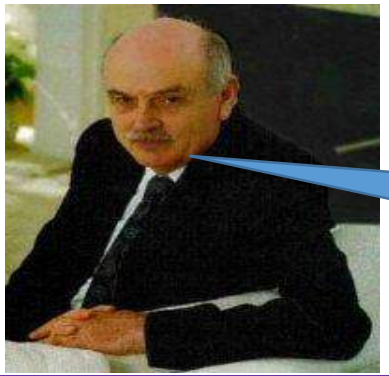
2nd Part of the Model: Querying

```
SELECT S.name
FROM Students S
WHERE S.gpa > 3.5;
```

We don't tell the system *how* or *where* to get the data- **just what we want, i.e., Querying is *declarative***

"Find names of all students with GPA > 3.5"

To make this happen, we need to translate the *declarative* query into a series of operators… we'll see this next!



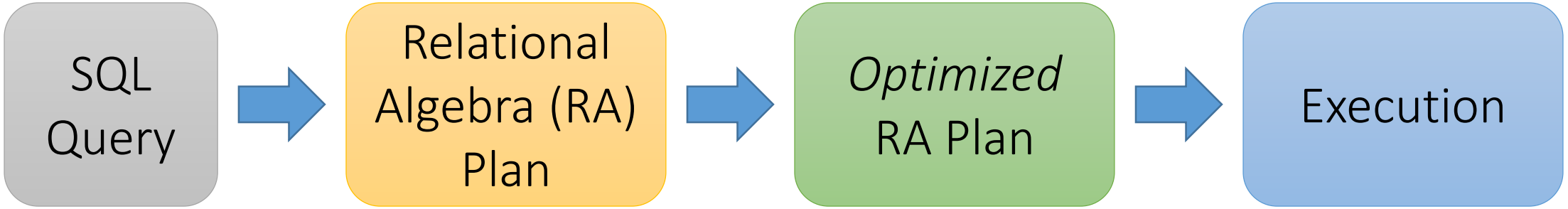
Actually, I showed how to do this translation for a much richer language!

Virtues of the model

- Physical independence (logical too), Declarative
- Simple, elegant, clean: Everything is a relation
- Why did it take multiple years?
  - Doubted it could be done *efficiently*.

RDBMS Architecture

How does an SQL engine work ?



Declarative query (from user)

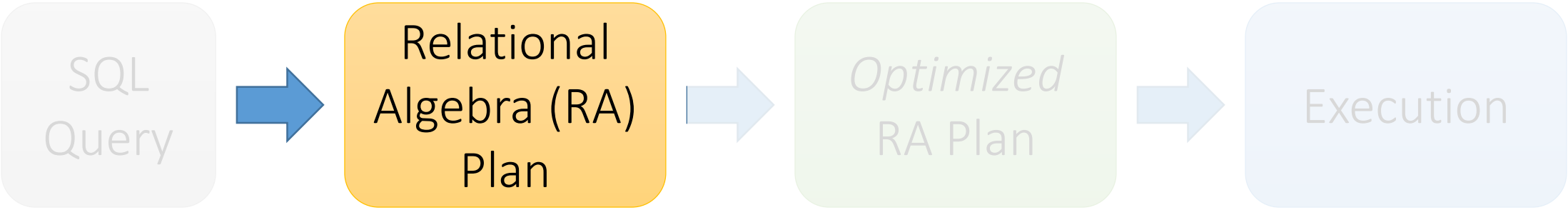
Translate to relational algebra expresson

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

RDBMS Architecture

How does a SQL engine work ?



Relational Algebra allows us to translate declarative (SQL) queries into precise and optimizable expressions!

Relational Algebra (RA)

- <u>Five **basic** operators:</u>
  - Selection: σ
  - Projection: Π
  - Cartesian Product: $\times$
  - Union: $\cup$
  - Difference: -

- <u>Derived or auxiliary operators:</u>
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ
  - Division

Keep in mind: RA operates on sets!

- RDBMSs use *multisets*, however in relational algebra formalism we will consider **sets!**

- Also: we will consider the ***named perspective***, where every attribute must have a unique name
  - →attribute order does not matter…

Now on to the basic RA operators…

Selection (σ)

- Returns all tuples which satisfy a condition
- Notation: $\sigma_c(R)$
- Examples
  - $\sigma_{Salary > 40000}$ (Employee)
  - $\sigma_{name = \text{“Smith”}}$ (Employee)
- The condition c can be
  - =, <, ≤, >, ≥, <>

Students(sid,sname,gpa)

SQL:

```
SELECT *
FROM Students
WHERE gpa > 3.5;
```

RA:

$$\sigma_{gpa > 3.5}(Students)$$

Another example:

SSN	Name	Salary
1234545	John	200000
5423341	Smith	600000
4352342	Fred	500000

$\sigma_{\text{Salary} > 40000}$ (Employee)

SSN	Name	Salary
5423341	Smith	600000
4352342	Fred	500000

Projection (Π)

- Eliminates columns, then removes duplicates
- Notation: $\Pi_{A1,\ldots,An}(R)$
- Example: project social-security number and names:
  - $\Pi_{SSN,\ Name}$ (Employee)
  - Output schema: *Answer (SSN, Name)*

Students(sid,sname,gpa)

SQL:

```
SELECT DISTINCT
  sname,
  gpa
FROM Students;
```

RA:

$$\Pi_{sname,gpa}(Students)$$

Another example:

SSN	Name	Salary
1234545	John	200000
5423341	John	600000
4352342	John	200000

$\Pi_{\text{Name,Salary}}$ (Employee)

Name	Salary
John	200000
John	600000

Note that RA Operators are Compositional!

Students(sid,sname,gpa)

```
SELECT DISTINCT
  sname,
  gpa
FROM Students
WHERE gpa > 3.5;
```

$$\Pi_{sname,gpa}(\sigma_{gpa>3.5}(Students))$$

$$\Updownarrow$$

$$\sigma_{gpa>3.5}(\Pi_{sname,gpa}(Students))$$

How do we represent this query in RA?

Are these logically equivalent?

Cross-Product (×)

- Each tuple in R1 with each tuple in R2
- Notation: R1 × R2
- Example:
  - Employee × Departments
- Mainly used to express joins

Students(sid,sname,gpa)
People(ssn,pname,address)

SQL:

```
SELECT *
FROM Students, People;
```

RA:

$$Students \times People$$

Another example:

People

ssn	pname	address
1234545	John	216 Rosse
5423341	Bob	217 Rosse

$\times$

Students

sid	sname	gpa
001	John	3.4
002	Bob	1.3

$$Students \times People$$

ssn	pname	address	sid	sname	gpa
1234545	John	216 Rosse	001	John	3.4
5423341	Bob	217 Rosse	001	John	3.4
1234545	John	216 Rosse	002	Bob	1.3
5423341	Bob	216 Rosse	002	Bob	1.3

Renaming (ρ – *Rho*)

- Changes the schema, not the instance
- A 'special' operator- neither basic nor derived
- Notation: $\rho_{B1,\ldots,Bn}(R)$

- **Note: this is shorthand for the proper form (since names, not order matters!):**
  - $\rho_{A1 \rightarrow B1,\ldots,An \rightarrow Bn}(R)$

Students(sid,sname,gpa)

SQL:

```
SELECT
  sid AS studId,
  sname AS name,
  gpa AS gradePtAvg
FROM Students;
```

RA:

$$\rho_{studId,name,gradePtAvg}(Students)$$

We care about this operator *because* we are working in a *named perspective*

Another example:

Students

sid	sname	gpa
001	John	3.4
002	Bob	1.3

$$\rho_{studId,name,gradePtAvg}(Students)$$

Students

studId	name	gradePtAvg
001	John	3.4
002	Bob	1.3

Natural Join (⋈)

- Notation: $R_1 \bowtie R_2$

- Joins R_1 and R_2 on *equality of all shared attributes*
  - If R_1 has attribute set A, and R_2 has attribute set B, and they share attributes $A \cap B = C$, can also be written: $R_1 \bowtie_C R_2$

- Our first example of a *derived* RA operator:
  - Meaning: $R_1 \bowtie R_2 = \Pi_{A \cup B}(\sigma_{C=D}(\rho_{C \to D}(R_1) \times R_2))$
  - Where:
    - The rename $\rho_{C \to D}$ renames the shared attributes in one of the relations
    - The selection $\sigma_{C=D}$ checks equality of the shared attributes
    - The projection $\Pi_{A \cup B}$ eliminates the duplicate common attributes

Students(sid,name,gpa)
People(ssn,name,address)

SQL:

```
SELECT DISTINCT
  ssid, S.name, gpa,
  ssn, address
FROM
  Students S,
  People P
WHERE S.name = P.name;
```

RA:

$$Students \bowtie People$$

Another example:

Students S

sid	S.name	gpa
001	John	3.4
002	Bob	1.3

$\bowtie$

People P

ssn	P.name	address
1234545	John	216 Rosse
5423341	Bob	217 Rosse

$$Students \bowtie People$$

sid	S.name	gpa	ssn	address
001	John	3.4	1234545	216 Rosse
002	Bob	1.3	5423341	217 Rosse

Natural Join

- Given schemas R(A, B, C, D), S(A, C, E), what is the schema of $R \bowtie S$?
- Given R(A, B, C), S(D, E), what is $R \bowtie S$?
- Given R(A, B), S(A, B), what is $R \bowtie S$?

Example: Converting SFW Query -> RA

Students(sid,sname,gpa)
People(ssn,sname,address)

```
SELECT DISTINCT
  gpa,
  address
FROM Students S,
     People P
WHERE gpa > 3.5 AND
   S.sname = P.sname;
```

$$\Pi_{gpa,address}(\sigma_{gpa>3.5}(S \bowtie P))$$

How do we represent this query in RA?

How would you find students who have taken all classes?

Students(studentname, coursecode)
Courses(coursecode)

Division

- Notation: $\boldsymbol{r \div s}$
- It has nothing to do with arithmetic division.
- Let *r* and *s* be relations on schemas *R* and *S* respectively where
  - $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
  - $S = (B_1, \ldots, B_n)$

  The result of r ÷ s is a relation on schema

  $R - S = (A_1, \ldots, A_m)$

  $$r \div s = \{ t \mid t \in \Pi_{R-S}(r) \wedge \forall u \in s (tu \in r) \}$$

  Where *tu* means the concatenation of tuples *t* and *u* to produce a single tuple

Database System Concepts, Silberschatz, Korth and Sudarshan

Division Operation - Example

- Relations r, s

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
ϵ	6
ϵ	1
β	2

r

B
1
2

s

- r ÷ s

A
α
β

Division Operation - Example

- Relations r, s

A	*B*	*C*	*D*	*E*
α	a	α	a	*1*
α	a	γ	a	*1*
α	a	γ	b	*1*
β	a	γ	a	*1*
β	a	γ	b	*3*
γ	a	γ	a	*1*
γ	a	γ	b	*1*
γ	a	β	b	*1*

r

D	*E*
a	*1*
b	*1*

s

- r ÷ s

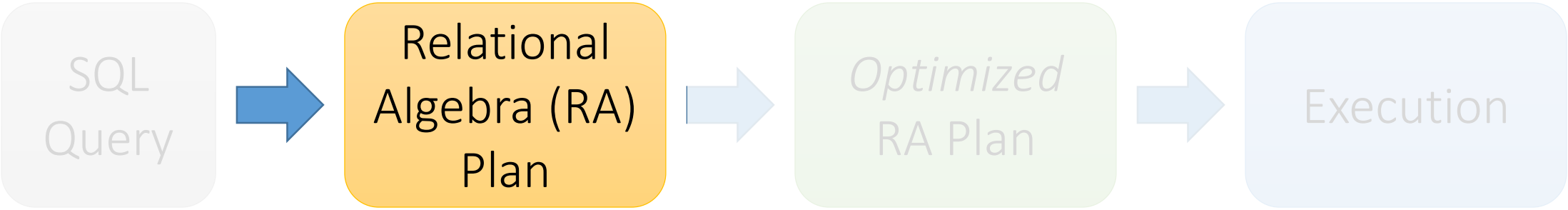
A	*B*	*C*
α	a	γ
γ	a	γ

Logical Equivalence of RA Plans

- Given relations R(A,B):
  - Here, projection & selection commute:
    - $\sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R))$
  - What about here?
    - $\sigma_{A=5}(\Pi_B(R)) = \Pi_B(\sigma_{A=5}(R)) \quad ???$

RDBMS Architecture

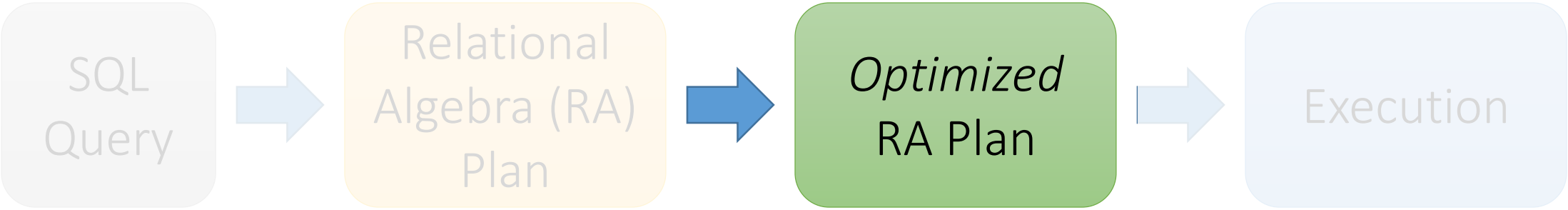
How does a SQL engine work ?



We saw how we can transform declarative SQL queries into precise, compositional RA plans

RDBMS Architecture

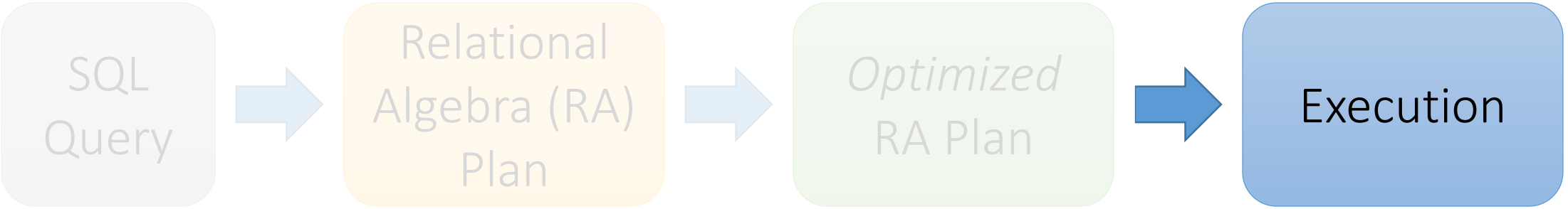
How does a SQL engine work ?



We'll look at how to then optimize these plans later!

RDBMS Architecture

How is the RA “plan” executed?



We will see later how to execute all the basic operators!

2. Advanced Relational Algebra

What you will learn about in this section

1. Set Operations in RA
2. Fancier RA
3. Extensions & Limitations

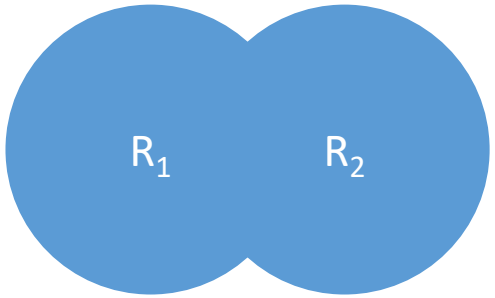
Relational Algebra (RA)

- <u>Five **basic** operators:</u>
  - Selection: σ
  - Projection: Π
  - Cartesian Product: $\times$
  - Union: $\cup$
  - Difference: -

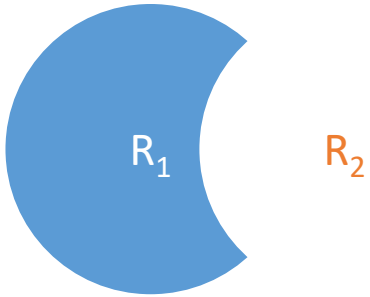
- <u>Derived or auxiliary operators:</u>
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ
  - Division

Union (∪) and Difference (–)

- R1 ∪ R2
- Example:
  - ActiveEmployees ∪ RetiredEmployees

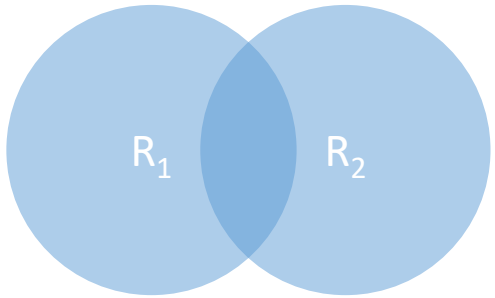


- R1 – R2
- Example:
  - AllEmployees – RetiredEmployees



What about Intersection (∩) ?

- It is a derived operator
- R1 ∩ R2 = R1 – (R1 – R2)
- Also expressed as a join!
- Example
  - UnionizedEmployees ∩ RetiredEmployees



Theta Join ($\bowtie_\theta$)

- A join that involves a predicate
- $R1 \bowtie_\theta R2 \;=\; \sigma_\theta (R1 \times R2)$
- Here θ can be any condition

Note that natural join is a theta join + a projection.

Students(sid,sname,gpa)
People(ssn,sname,address)

SQL:

```
SELECT *
FROM
  Students,People
WHERE θ;
```

RA:

$$Students \bowtie_\theta People$$

Equi-join ($\bowtie_{A=B}$)

- A theta join where θ is an equality
- $R1 \bowtie_{A=B} R2 = \sigma_{A=B} (R1 \times R2)$
- Example:
  - Employee $\bowtie_{SSN=SSN}$ Dependents

Most common join in practice!

Students(sid,sname,gpa)
People(ssn,pname,address)

SQL:

```
SELECT *
FROM
  Students S,
  People P
WHERE sname = pname;
```

RA:

$$S \bowtie_{sname=pname} P$$

Semijoin (⋉)

- $R \ltimes S = \Pi_{A1,\ldots,An} (R \bowtie S)$
- Where $A_1, \ldots, A_n$ are the attributes in R
- Example:
  - Employee ⋉ Dependents

Students(sid,sname,gpa)
People(ssn,pname,address)

SQL:

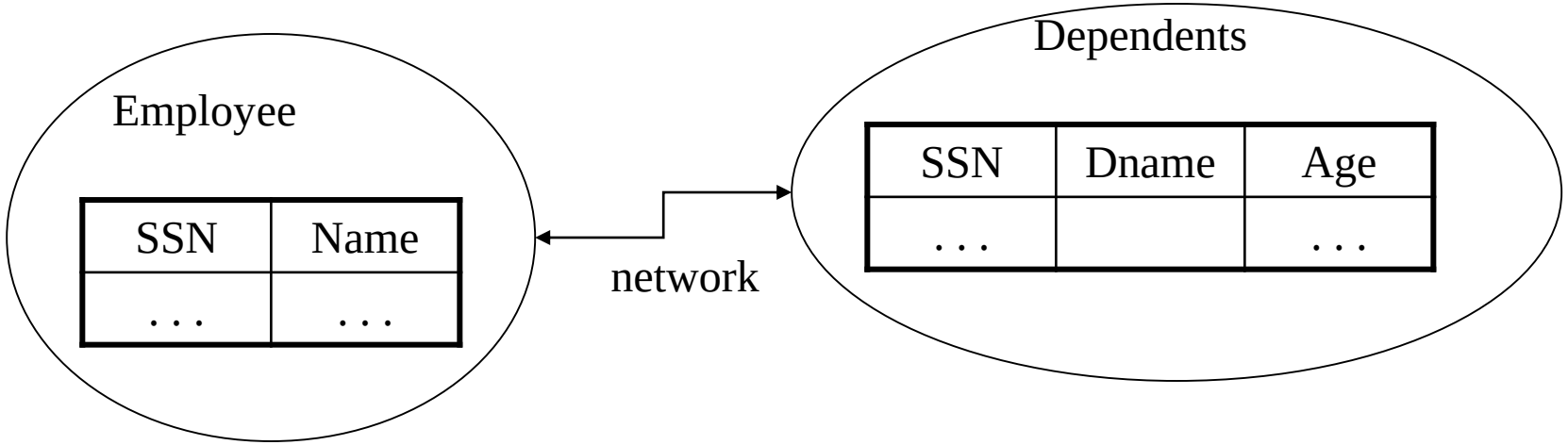
```
SELECT DISTINCT
  sid,sname,gpa
FROM
  Students,People
WHERE
  sname = pname;
```

RA:

$$Students \ltimes People$$

Semijoins in Distributed Databases

- Semijoins are often used to compute natural joins in distributed databases



$$\text{Employee} \bowtie_{ssn=ssn} (\sigma_{age>71} (\text{Dependents}))$$

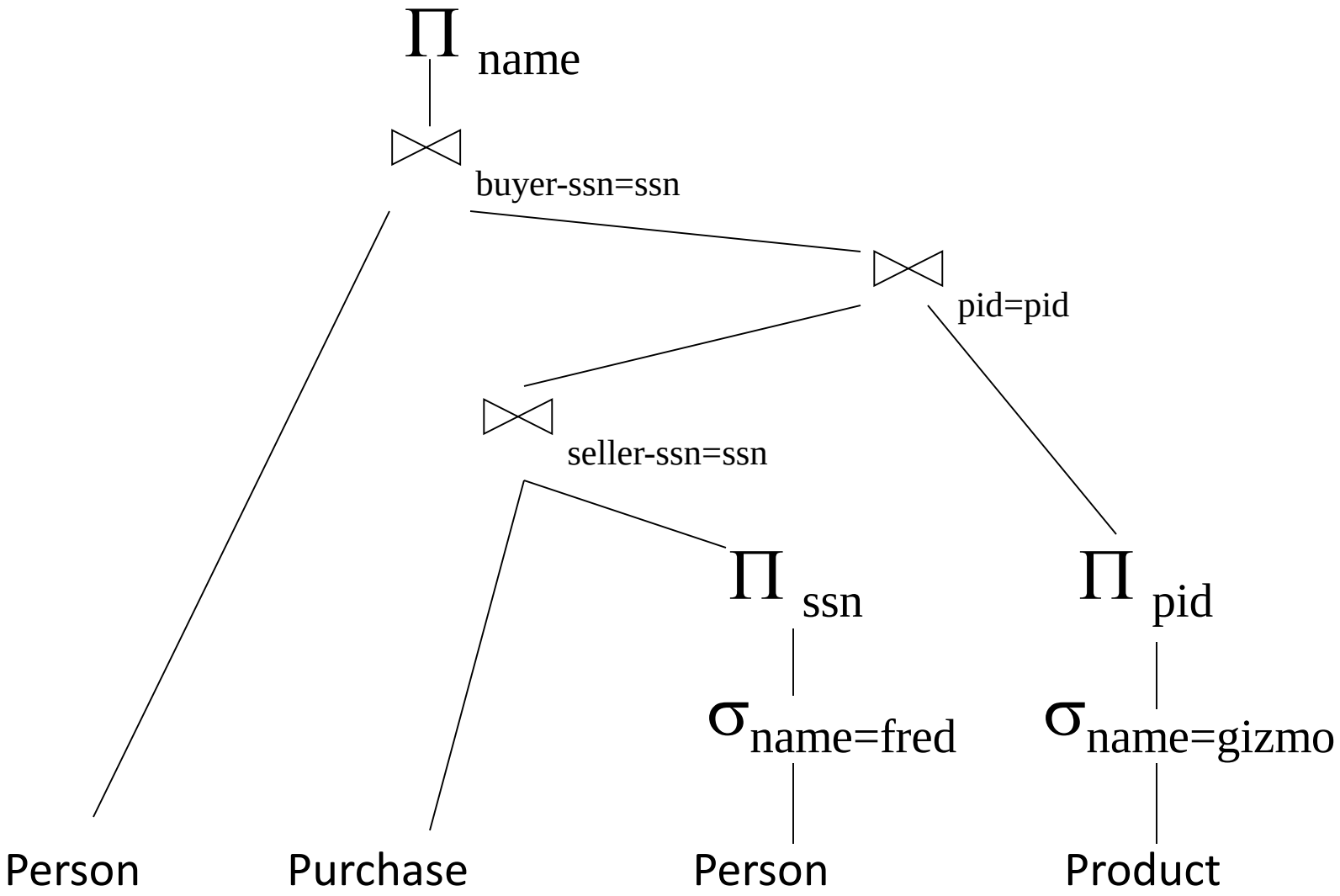
Send less data to reduce network bandwidth!

$$T = \Pi_{SSN}\, \sigma_{age>71} (\text{Dependents})$$

$$R = \text{Employee} \ltimes T$$

$$\text{Answer} = R \bowtie \text{Dependents}$$

RA Expressions Can Get Complex!



Operations on Multisets

All RA operations need to be defined carefully on bags

- $\sigma_C(R)$: preserve the number of occurrences
- $\Pi_A(R)$: no duplicate elimination
- Cross-product, join: no duplicate elimination

This is important- relational engines work on multisets, not sets!

RA has Limitations !

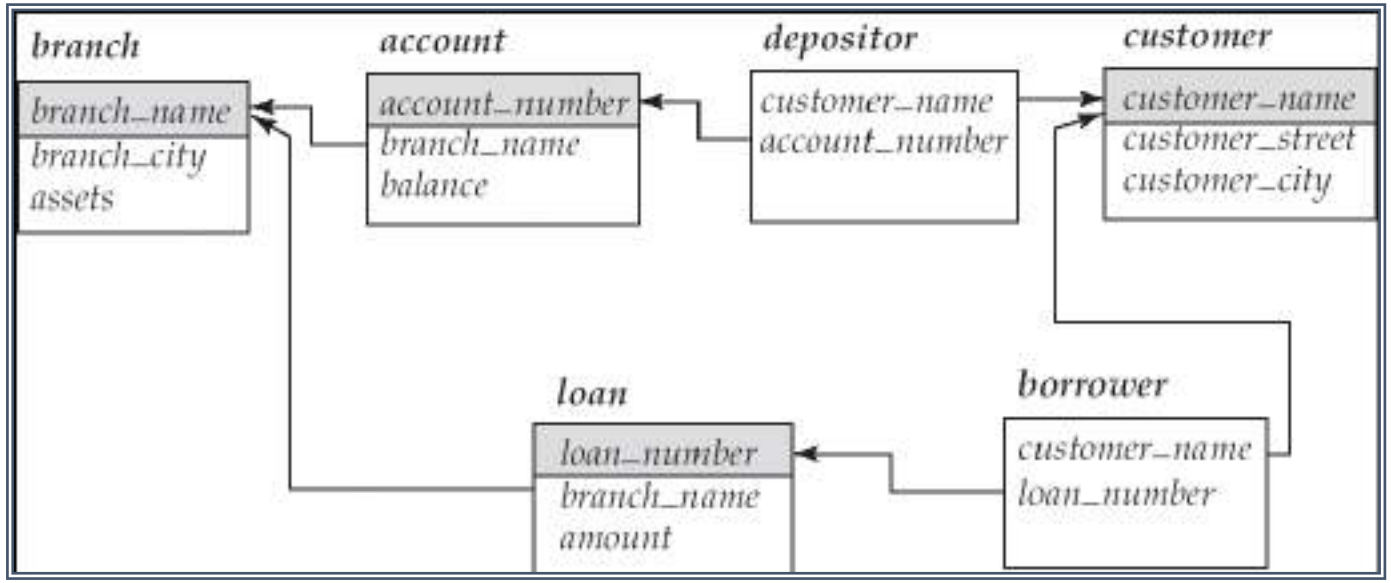
- Cannot compute “transitive closure”

Name1	Name2	Relationship
Fred	Mary	Father
Mary	Joe	Cousin
Mary	Bill	Spouse
Nancy	Lou	Sister

- Find all direct and indirect relatives of Fred
- Cannot express in RA !!!
  - Need to write C program, use a graph engine, or modern SQL…

Example: Banking Database

branch (branch_name, branch_city, assets)
customer (customer_name, customer_street, customer_city)
account (account_number, branch_name, balance)
loan (loan_number, branch_name, amount)
depositor (customer_name, account_number)
borrower (customer_name, loan_number)



Example Queries

- Find all loans of over $1200

$$\sigma_{amount > 1200}(loan)$$

- Find the loan number for each loan of an amount greater than $1200

$$\Pi_{loan_number}(\sigma_{amount > 1200}(loan))$$

Database System Concepts, Silberschatz, Korth and Sudarshan

Example Queries

- Find the names of all customers who have a loan, an account, or both, from the bank

$$\Pi_{customer_name}(borrower) \cup \Pi_{customer_name}(depositor)$$

- Find the names of all customers who have a loan and an account at bank.

$$\Pi_{customer_name}(borrower) \cap \Pi_{customer_name}(depositor)$$

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.

$$\Pi_{customer_name}(\sigma_{branch_name=\text{“Perryridge”}}$$
$$(\sigma_{borrower.loan_number\ =\ loan.loan_number}(borrower\ x\ loan)))$$

- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$$\Pi_{customer_name}(\sigma_{branch_name\ =\ \text{“Perryridge”}}$$

$$(\sigma_{borrower.loan_number\ =\ loan.loan_number}(\text{borrower x loan}))) - \Pi_{customer_name}(\text{depositor})$$

Database System Concepts, Silberschatz, Korth and Sudarshan

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
  - **Query 1**

$$\Pi_{customer_name} (\sigma_{branch_name = \text{“Perryridge”}} (\sigma_{borrower.loan_number = loan.loan_number} (borrower \times loan)))$$

  - **Query 2**

$$\Pi_{customer_name}(\sigma_{loan.loan_number = borrower.loan_number} ((\sigma_{branch_name = \text{“Perryridge”}} (loan)) \times borrower))$$

Database System Concepts, Silberschatz, Korth and Sudarshan

Example Queries

- Find all customers who have an account at all branches located in Brooklyn city.

$$\Pi_{customer_name,\ branch_name}(depositor \bowtie account) \div \Pi_{branch_name}(\sigma_{branch_city = \text{“Brooklyn”}}(branch))$$

Database System Concepts, Silberschatz, Korth and Sudarshan

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the *CS145 Introduction to Databases* lecture given by *Christopher Ré* at *Stanford University* (http://web.stanford.edu/class/cs145/).

Design Theory

BBM471 Database Management Systems

Design Theory I

Today's Lecture

1. Normal forms & functional dependencies
2. Finding functional dependencies
3. Closures, superkeys & keys

1. Normal forms & functional dependencies

What you will learn about in this section

1. Overview of design theory & normal forms
2. Data anomalies & constraints
3. Functional dependencies

Design Theory

- Design theory is about how to represent your data to avoid ***anomalies***.

- It is a mostly mechanical process
  - Tools can carry out routine portions

Normal Forms

- 1st Normal Form (1NF) = All tables are flat

- *2nd Normal Form = disused*

- **Boyce-Codd Normal Form (BCNF)**
- **3rd Normal Form (3NF)**

DB designs based on *functional dependencies*, intended to prevent data ***anomalies***

Our focus in this lecture

- *4th and 5th Normal Forms = see text books*

1st Normal Form (1NF)

Student	Courses
Mary	{CS145,CS229}
Joe	{CS145,CS106}
…	…

Violates 1NF.

Student	Courses
Mary	CS145
Mary	CS229
Joe	CS145
Joe	CS106

In 1st NF

1NF Constraint: Types must be atomic!

Data Anomalies & Constraints

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

If every course is in only one room, contains ***redundant*** information!

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
Mary	CS145	B01
Joe	CS145	C12
Sam	CS145	B01
..	..	..

If we update the room number for one tuple, we get inconsistent data = an ***update*** **anomaly**

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
..	..	..

If everyone drops the class, we lose what room the class is in! = a ***delete*** **anomaly**

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

CS229	C12

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Similarly, we can't reserve a room without students = an ***insert*** **anomaly**

Constraints Prevent (some) Anomalies in the Data

Student	Course
Mary	CS145
Joe	CS145
Sam	CS145
..	..

Course	Room
CS145	B01
CS229	C12

Is this form better?

- Redundancy?
- Update anomaly?
- Delete anomaly?
- Insert anomaly?

Today: develop theory to understand why this design may be better **and** how to find this *decomposition*…

Functional Dependencies

Functional Dependency

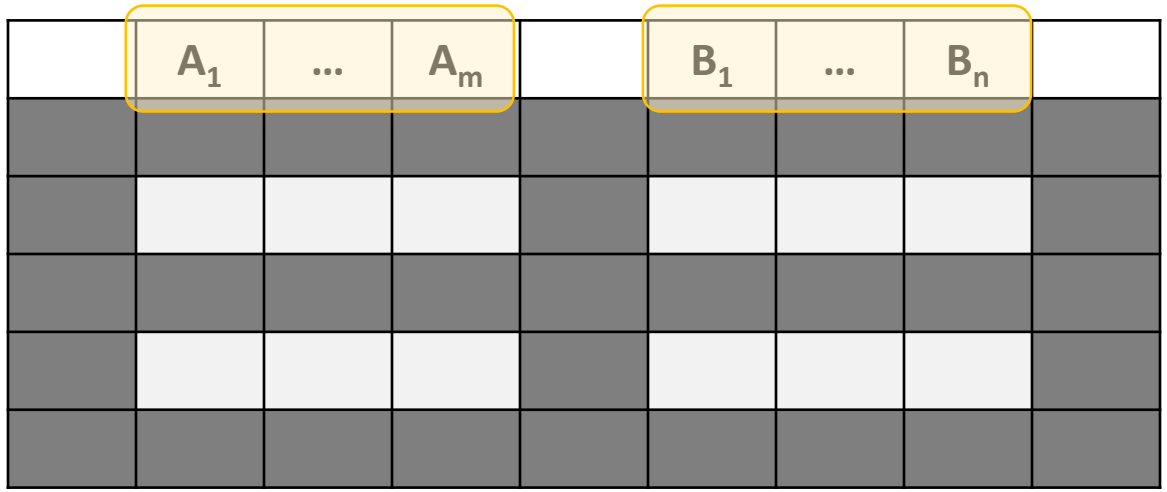
Def: Let A,B be *sets* of attributes
We write A → B or say A ***functionally determines*** B
if, for any tuples t_1 and t_2:

$$t_1[A] = t_2[A] \text{ implies } t_1[B] = t_2[B]$$

and we call A → B a **functional dependency**

In another words: A->B means that
"whenever two tuples agree on A then they agree on B."

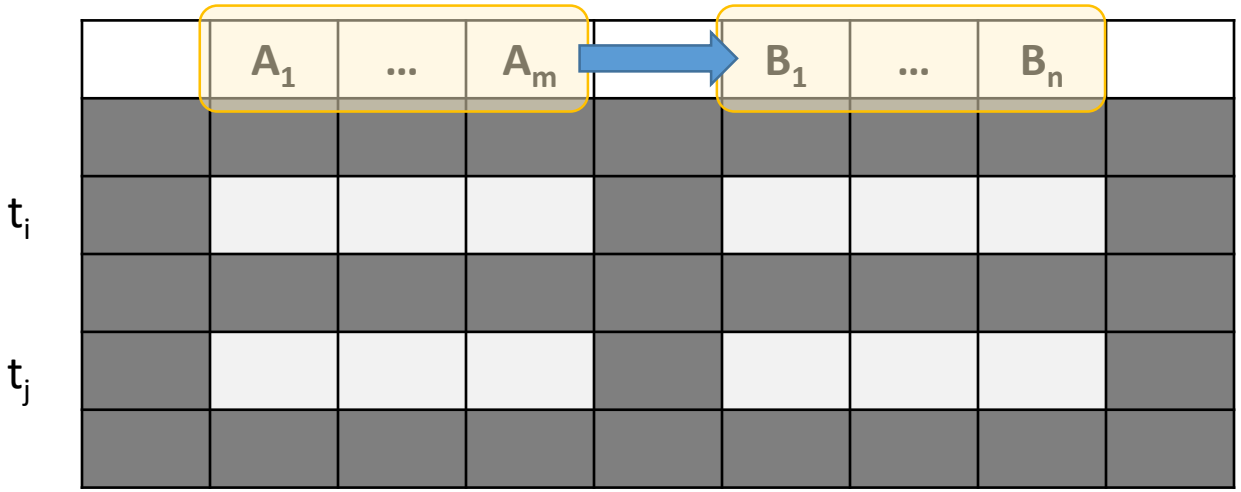
A Picture Of FDs



<u>Defn (again):</u>

Given attribute sets **A**={A_1,…,A_m} and **B** = {B_1,…B_n} in **R**,

A Picture Of FDs

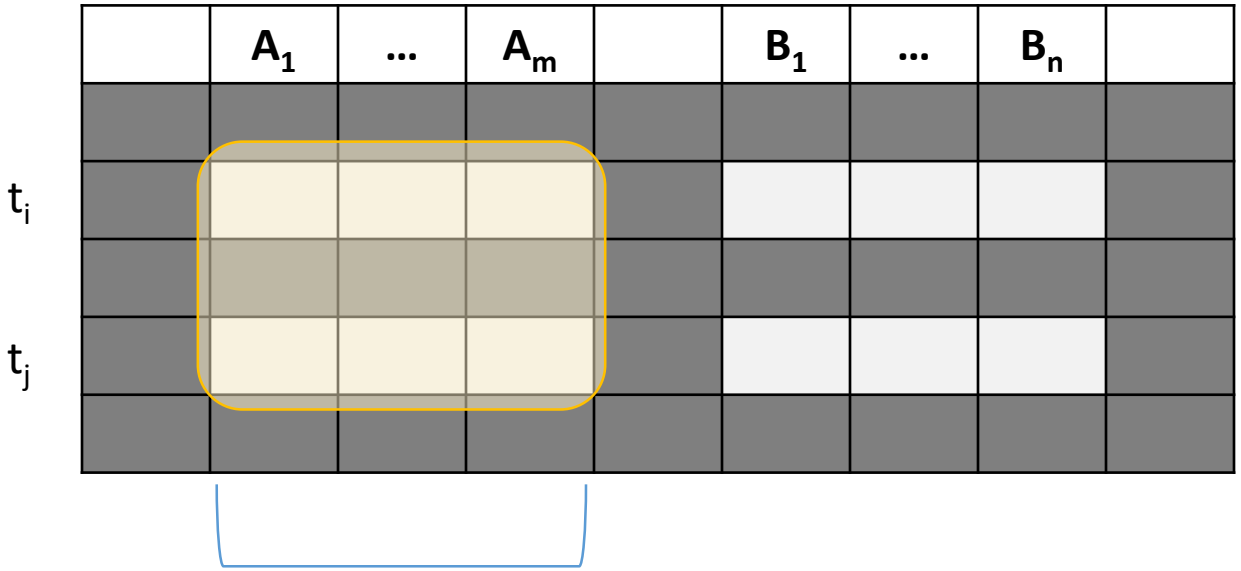


<u>Defn (again):</u>

Given attribute sets **A={A_1,…,A_m}** and **B = {B_1,…B_n}** in **R,**

The ***functional dependency*** **A→ B on R** holds if for ***any*** t_i,t_j in R:

A Picture Of FDs



If t_i, t_j agree here..

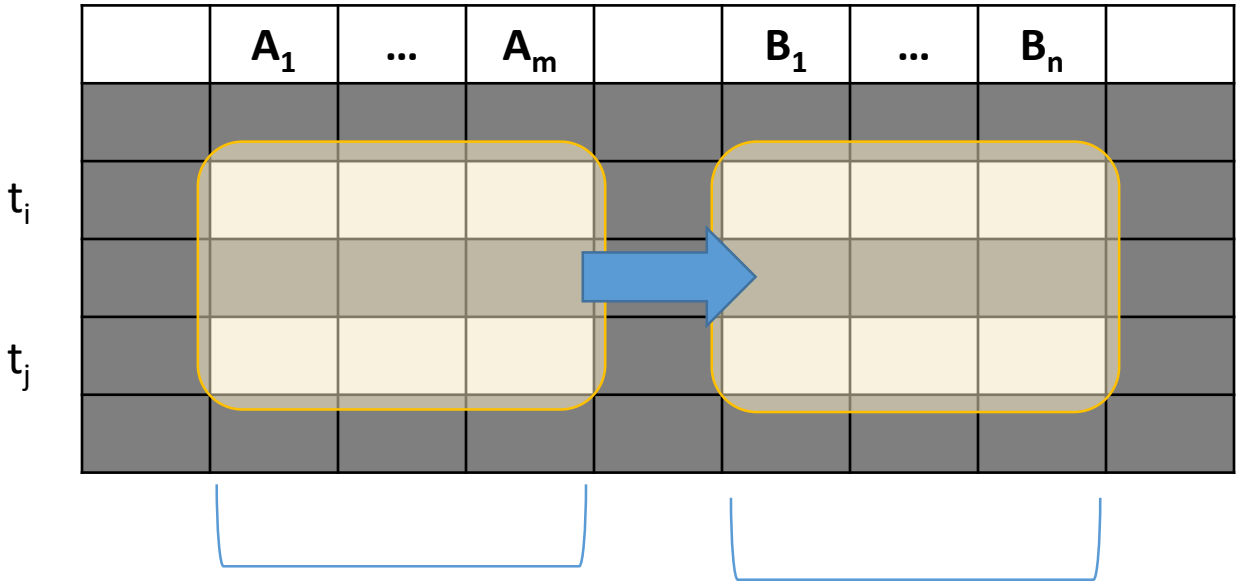
<u>Defn (again):</u>

Given attribute sets **A={A_1,…,A_m}** and **B = {B_1,…B_n}** in **R,**

The ***functional dependency*** **A→ B on R** holds if for ***any*** t_i,t_j in R:

$t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

A Picture Of FDs



If t_i,t_j agree here..

…they also agree here!

<u>Defn (again):</u>

Given attribute sets **A={A_1,…,A_m}** and **B = {B_1,…B_n}** in **R,**

The ***functional dependency*** **A→ B on R** holds if for ***any*** t_i,t_j in R:

<u>**if**</u> $t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

<u>**then**</u> $t_i[B_1] = t_j[B_1]$ AND $t_i[B_2]=t_j[B_2]$ AND … AND $t_i[B_n] = t_j[B_n]$

FDs for Relational Schema Design

- High-level idea: **why do we care about FDs?**

1. Start with some relational *schema*
2. Model its *functional dependencies (FDs)*
3. Use these to *design a better schema*
   - One which minimizes the possibility of anomalies

Functional Dependencies as Constraints

A **functional dependency** is a form of **constraint**

- *Holds* on some instances not others.
- Part of the schema, helps define a valid *instance*.

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Note: The FD {Course} -> {Room} ***holds on this instance***

Recall: an ***instance*** *of a schema is a multiset of tuples conforming to that schema,* ***i.e. a table***

Functional Dependencies as Constraints

Note that:

- You can check if an FD is **violated** by examining a single instance;
- However, you **cannot prove** that an FD is part of the schema by examining a single instance.
  - *This would require checking every valid instance*

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

However, cannot *prove* that the FD {Course} -> {Room} is ***part of the schema***

More Examples

An FD is a constraint which holds, or does not hold on an instance:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

More Examples

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876 ←	Salesrep
E1111	Smith	9876 ←	Salesrep
E9999	Mary	1234	Lawyer

{Position} → {Phone}

More Examples

EmpID	Name	Phone	Position
E0045	Smith	1234 →	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234 →	Lawyer

but *not* {Phone} → {Position}

2. Finding functional dependencies

What you will learn about in this section

1. “Good” vs. “Bad” FDs: Intuition
2. Finding FDs
3. Closures

“Good” vs. “Bad” FDs

We can start to develop a notion of **good** vs. **bad** FDs:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

<u>Intuitively:</u>

EmpID -> Name, Phone, Position is *“good FD”*

- ***Minimal redundancy, less possibility of anomalies***

“Good” vs. “Bad” FDs

We can start to develop a notion of **good** vs. **bad** FDs:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

<u>Intuitively:</u>

EmpID -> Name, Phone, Position is *“good FD”*

Position -> Phone *is a “bad FD”*

- ***Redundancy! Possibility of data anomalies***

“Good” vs. “Bad” FDs

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Returning to our original example… can you see how the “bad FD” {Course} -> {Room} could lead to an:

- Update Anomaly
- Insert Anomaly
- Delete Anomaly
- …

Given a set of FDs (from user) our goal is to:

1. **Find all FDs, and**
2. **Eliminate the “Bad Ones".**

FDs for Relational Schema Design

- High-level idea: **why do we care about FDs?**

1. Start with some relational *schema*
2. Find out its *functional dependencies (FDs)*
3. Use these to *design a better schema*
   - One which minimizes possibility of anomalies

This part can be tricky!

Finding Functional Dependencies

- There can be a very **large number** of FDs…
  - *How to find them all efficiently?*

- We can't necessarily show that any FD will hold **on all instances…**
  - *How to do this?*

We will start with this problem:
Given a set of FDs, F, what other FDs ***must*** hold?

Finding Functional Dependencies

Equivalent to asking: Given a set of FDs, $F = \{f_1,\ldots f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Finding Functional Dependencies

Example:

Products

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Department}
3. {Color, Category} → {Price}

Given the provided FDs, we can see that **{Name, Category} → {Price}** must also hold on **any instance**…

Which / how many other FDs do?!?

Finding Functional Dependencies

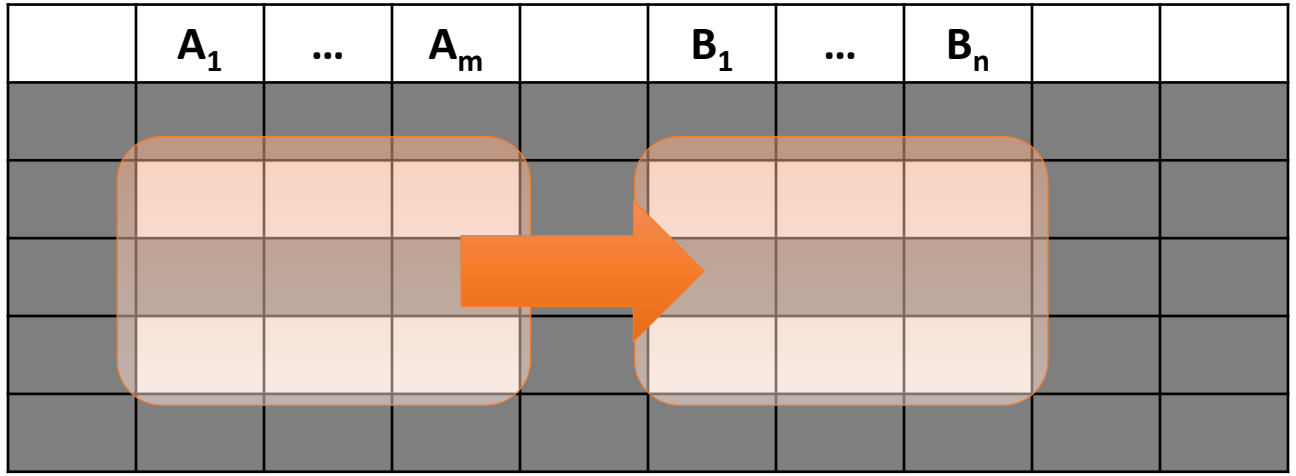
Equivalent to asking: Given a set of FDs, F = $\{f_1,\ldots f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Answer: Three simple rules called **Armstrong's Rules.**

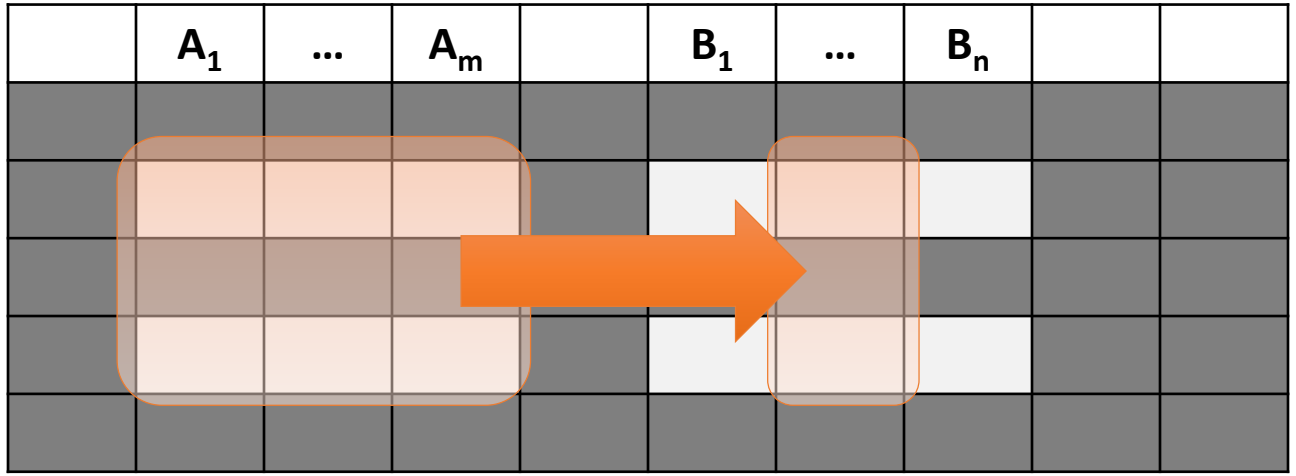
1. **Split/Combine,**
2. **Reduction (Trivial), and**
3. **Transitivity…** *ideas by picture*

1. Split/Combine



$A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n$

1. Split/Combine

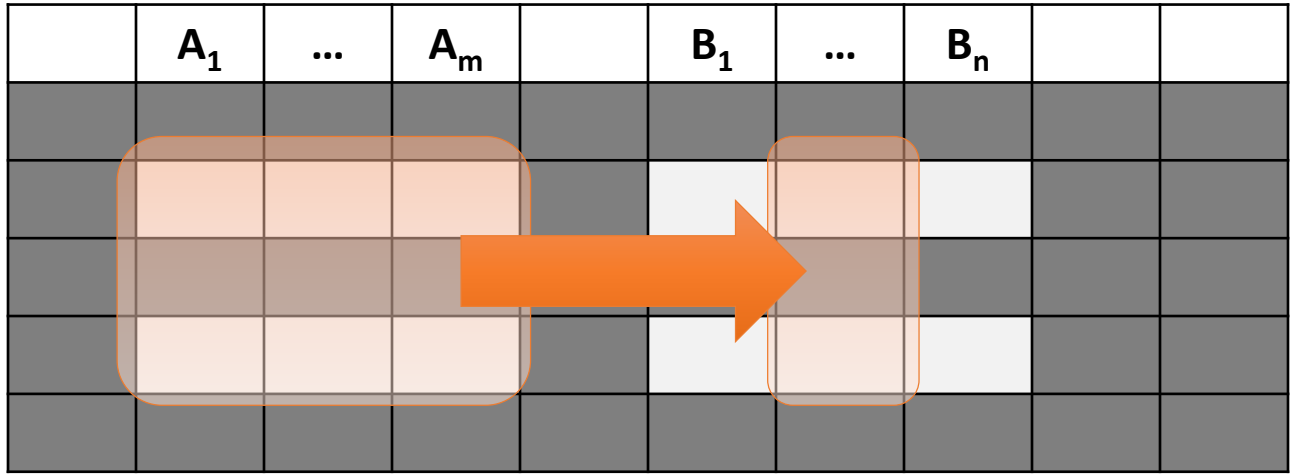


$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

… is equivalent to the following *n* FDs…

$A_1,\ldots,A_m \rightarrow B_i$ for i=1,…,n

1. Split/Combine

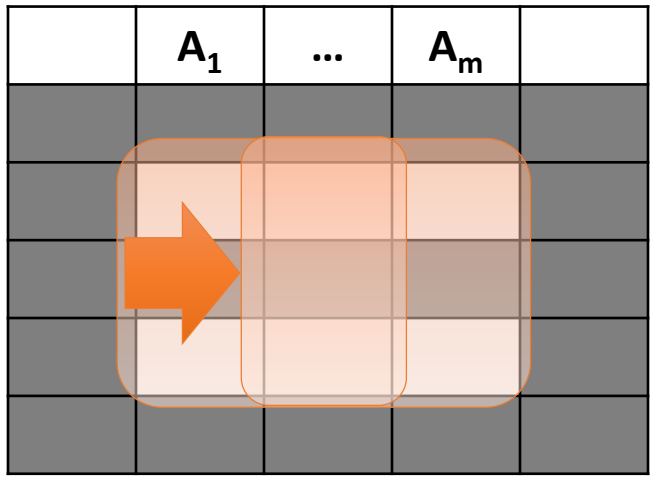


And vice-versa, $A_1,\ldots,A_m \rightarrow B_i$ for $i=1,\ldots,n$

… is equivalent to …

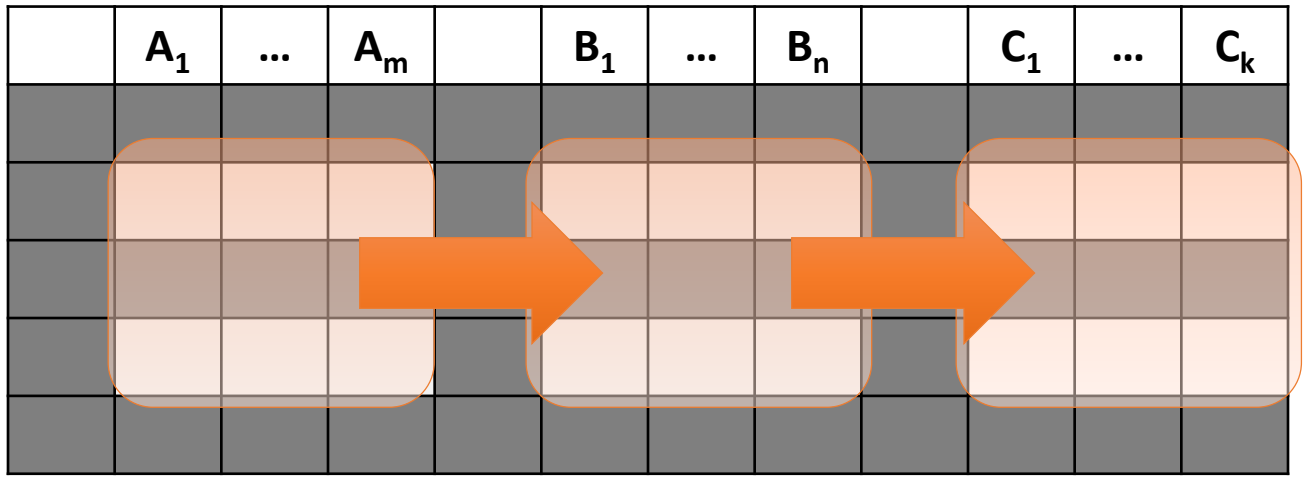
$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

Reduction/Trivial



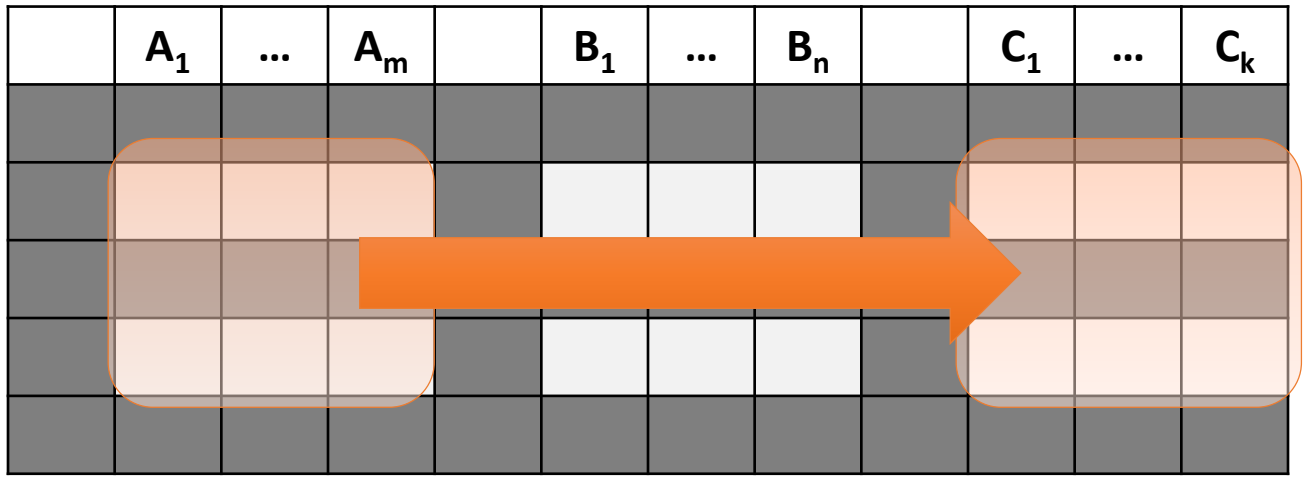
$A_1,\ldots,A_m \rightarrow A_j$ for any $j=1,\ldots,m$

3. Transitive Closure



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and
$B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

3. Transitive Closure



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and
$B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

implies

$A_1,\ldots,A_m \rightarrow C_1,\ldots,C_k$

Finding Functional Dependencies

Example:

Products

Name	Color	Category	Dept	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept}
3. {Color, Category} → {Price}

Which / how many other FDs hold?

Finding Functional Dependencies

Example:

Inferred FDs:

Inferred FD	Rule used
4. {Name, Category} -> {Name}	?
5. {Name, Category} -> {Color}	?
6. {Name, Category} -> {Category}	?
7. {Name, Category -> {Color, Category}	?
8. {Name, Category} -> {Price}	?

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

Which / how many other FDs hold?

Finding Functional Dependencies

Example:

Inferred FDs:

Inferred FD	Rule used
4. {Name, Category} -> {Name}	Trivial
5. {Name, Category} -> {Color}	Transitive (4 -> 1)
6. {Name, Category} -> {Category}	Trivial
7. {Name, Category -> {Color, Category}	Split/combine (5 + 6)
8. {Name, Category} -> {Price}	Transitive (7 -> 3)

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

Can we find an algorithmic way to do this?

Closures

Closure of a set of Attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs **F:**
Then the <u>closure</u>, $\{A_1, \ldots, A_n\}^+$ is the set of attributes **B** s.t. $\{A_1, \ldots, A_n\} \rightarrow B$

<u>Example:</u> F =

{name} → {color}
{category} → {dept}
{color, category} → {price}

Example Closures:

$\{name\}^+$ = {name, color}
$\{name, category\}^+$ =
{name, category, color, dept, price}
$\{color\}^+$ = {color}

Closure Algorithm

Start with $X = \{A_1, \ldots, A_n\}$ and set of FDs F.

Repeat until X doesn't change; **do**:

if $\{B_1, \ldots, B_n\} \rightarrow C$ is entailed by F

and $\{B_1, \ldots, B_n\} \subseteq X$

then add C to X.

Return X as X^+

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

$\{name, category\}^+$ =
{name, category}

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

$\{name, category\}^+$ =
{name, category}

$\{name, category\}^+$ =
{name, category, color}

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

$\{name, category\}^+$ =
{name, category}

$\{name, category\}^+$ =
{name, category, color}

$\{name, category\}^+$ =
{name, category, color, dept}

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

$\{name, category\}^+$ =
{name, category}

$\{name, category\}^+$ =
{name, category, color}

$\{name, category\}^+$ =
{name, category, color, dept}

$\{name, category\}^+$ =
{name, category, color, dept, price}

Example

R(A,B,C,D,E,F)

$\{A,B\} \rightarrow \{C\}$
$\{A,D\} \rightarrow \{E\}$
$\{B\} \rightarrow \{D\}$
$\{A,F\} \rightarrow \{B\}$

Compute $\{A,B\}^+ = \{A, B, \qquad\qquad\qquad\qquad \}$

Compute $\{A, F\}^+ = \{A, F, \qquad\qquad\qquad\qquad \}$

Example

R(A,B,C,D,E,F)

$\{A,B\} \rightarrow \{C\}$
$\{A,D\} \rightarrow \{E\}$
$\{B\} \rightarrow \{D\}$
$\{A,F\} \rightarrow \{B\}$

Compute $\{A,B\}^+ = \{A, B, C, D \quad \}$

Compute $\{A, F\}^+ = \{A, F, B \quad \}$

Example

R(A,B,C,D,E,F)

$\{A,B\} \rightarrow \{C\}$
$\{A,D\} \rightarrow \{E\}$
$\{B\} \rightarrow \{D\}$
$\{A,F\} \rightarrow \{B\}$

Compute $\{A, B\}^+ = \{A, B, C, D, E\}$

Compute $\{A, F\}^+ = \{A, B, C, D, E, F\}$

3. Closures, Superkeys & Keys

What you will learn about in this section

1. Closures Pt. II

2. Superkeys & Keys

Why Do We Need the Closure?

- With closure we can find all FD's easily

- To check if $X \rightarrow A$

  1. Compute X^+

  2. Check if $A \in X^+$

Note here that **X** is a *set* of attributes, but **A** is a *single* attribute. Why does considering FDs of this form suffice?

Recall the <u>**Split/combine**</u> rule:
$X \rightarrow A_1, \ldots, X \rightarrow A_n$
implies
$X \rightarrow \{A_1, \ldots, A_n\}$

Using Closure to Infer ALL FDs

Example:

Given F =

$\{A,B\} \rightarrow C$
$\{A,D\} \rightarrow B$
$\{B\} \rightarrow D$

Step 1: Compute X^+, for every set of attributes X:

$\{A\}^+ = \{A\}$
$\{B\}^+ = \{B,D\}$
$\{C\}^+ = \{C\}$
$\{D\}^+ = \{D\}$
$\{A,B\}^+ = \{A,B,C,D\}$
$\{A,C\}^+ = \{A,C\}$
$\{A,D\}^+ = \{A,B,C,D\}$
$\{A,B,C\}^+ = \{A,B,D\}^+ = \{A,C,D\}^+ = \{A,B,C,D\}$ $\{B,C,D\}^+ = \{B,C,D\}$
$\{A,B,C,D\}^+ = \{A,B,C,D\}$

Using Closure to Infer ALL FDs

Example:

Given F =

$\{A,B\} \rightarrow C$
$\{A,D\} \rightarrow B$
$\{B\} \rightarrow D$

Step 1: Compute X^+, for every set of attributes X:

$\{A\}^+ = \{A\}$, $\{B\}^+ = \{B,D\}$, $\{C\}^+ = \{C\}$, $\{D\}^+ = \{D\}$, $\{A,B\}^+ = \{A,B,C,D\}$, $\{A,C\}^+ = \{A,C\}$, $\{A,D\}^+ = \{A,B,C,D\}$, $\{A,B,C\}^+ = \{A,B,D\}^+ = \{A,C,D\}^+ = \{A,B,C,D\}$, $\{B,C,D\}^+ = \{B,C,D\}$, $\{A,B,C,D\}^+ = \{A,B,C,D\}$

Step 2: Enumerate all FDs $X \rightarrow Y$, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

$\{A,B\} \rightarrow \{C,D\}$, $\{A,D\} \rightarrow \{B,C\}$,
$\{A,B,C\} \rightarrow \{D\}$, $\{A,B,D\} \rightarrow \{C\}$,
$\{A,C,D\} \rightarrow \{B\}$

Using Closure to Infer ALL FDs

Example:

Given F =

$\{A,B\} \rightarrow C$
$\{A,D\} \rightarrow B$
$\{B\} \rightarrow D$

Step 1: Compute X^+, for every set of attributes X:

$\{A\}^+ = \{A\}$, $\{B\}^+ = \{B,D\}$, $\{C\}^+ = \{C\}$, $\{D\}^+ = \{D\}$, $\{A,B\}^+ = \{A,B,C,D\}$, $\{A,C\}^+ = \{A,C\}$, $\{A,D\}^+ = \{A,B,C,D\}$, $\{A,B,C\}^+ = \{A,B,D\}^+ = \{A,C,D\}^+ = \{A,B,C,D\}$, $\{B,C,D\}^+ = \{B,C,D\}$, $\{A,B,C,D\}^+ = \{A,B,C,D\}$

Step 2: Enumerate all FDs $X \rightarrow Y$, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

"Y is in the closure of X"

$\{A,B\} \rightarrow \{C,D\}$, $\{A,D\} \rightarrow \{B,C\}$,
$\{A,B,C\} \rightarrow \{D\}$, $\{A,B,D\} \rightarrow \{C\}$,
$\{A,C,D\} \rightarrow \{B\}$

Using Closure to Infer ALL FDs

Example:

Given F =

$\{A,B\} \rightarrow C$
$\{A,D\} \rightarrow B$
$\{B\} \rightarrow D$

Step 1: Compute X^+, for every set of attributes X:

$\{A\}^+ = \{A\}$, $\{B\}^+ = \{B,D\}$, $\{C\}^+ = \{C\}$, $\{D\}^+ = \{D\}$, $\{A,B\}^+ = \{A,B,C,D\}$, $\{A,C\}^+ = \{A,C\}$, $\{A,D\}^+ = \{A,B,C,D\}$, $\{A,B,C\}^+ = \{A,B,D\}^+ = \{A,C,D\}^+ = \{A,B,C,D\}$, $\{B,C,D\}^+ = \{B,C,D\}$, $\{A,B,C,D\}^+ = \{A,B,C,D\}$

Step 2: Enumerate all FDs $X \rightarrow Y$, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

The FD $X \rightarrow Y$ is non-trivial

$\{A,B\} \rightarrow \{C,D\}$, $\{A,D\} \rightarrow \{B,C\}$,
$\{A,B,C\} \rightarrow \{D\}$, $\{A,B,D\} \rightarrow \{C\}$,
$\{A,C,D\} \rightarrow \{B\}$

Superkeys and Keys

Keys and Superkeys

A **superkey** is a set of attributes $A_1, \ldots, A_n$ s.t. for *any other* attribute **B** in R, we have $\{A_1, \ldots, A_n\} \rightarrow B$

i.e. all attributes are *functionally determined* by a superkey.

Note: superkey is short for super set of keys.

A **key** is a *minimal* superkey

Meaning that no subset of a key is also a superkey

Finding Keys and Superkeys

- For each set of attributes X

  1. Compute X^+

  2. If X^+ = set of all attributes then X is a **superkey**

  3. If X is minimal, then it is a **key**

Do we need to check all sets of attributes? Which sets?

Example of Finding Keys

Product(name, price, category, color)

{name, category} → price
{category} → color

What is a key?

Example of Keys

Product(name, price, category, color)

{name, category} → price
{category} → color

$\{name, category\}^+$ = {name, price, category, color}

= the set of all attributes

⟹ this is a **superkey**

⟹ this is a **key**, since neither name nor category alone is a superkey

Design Theory II

Today's Lecture

1. Boyce-Codd Normal Form
2. Decompositions & 3NF
3. MVDs

1. Boyce-Codd Normal Form

What you will learn about in this section

1. Conceptual Design
2. Boyce-Codd Normal Form
3. The BCNF Decomposition Algorithm

Conceptual Design

Back to Conceptual Design

Now that we know how to find FDs, it's a straight-forward process:

1. Search for "bad" FDs
2. If there are any, then *keep decomposing the table into sub-tables* until no more bad FDs
3. When done, the database schema is *normalized*

Prime vs Non-Prime Attributes

- **Prime Attributes**
  - Attribute set that belongs to any candidate key are called Prime Attributes.
- **Non-Prime Attribute**
  - Attribute set does not belong to any candidate key are called Non-Prime Attributes.

Normal Forms

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce Codd Normal Form (BCNF)

First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single (atomic) valued attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.

First Normal Form (1NF) ?

id	name	course
1	Fuat	BBM471 BBM467 CMP652
2	Zeynep	BBM101 BBM102

UNF – Unnormalized Form
or, NF2

Second Normal Form (2NF)

For a table to be in the Second Normal Form

1. It should be in the First Normal form.
2. No non-prime attribute is dependent on any proper subset of any candidate key of the relation, i.e., *no partial dependency*.
   - All non-prime attributes are fully functional dependent on the primary key

Second Normal Form (2NF) ?

id	name	course	course name
1	Fuat	BBM471	Database Management Systems
1	Fuat	BBM467	Data Intensive Apps
1	Fuat	CMP652	Next Gen DBMS
2	Zeynep	BBM101	Programming I
2	Zeynep	BBM102	Programming II

Second Normal Form (2NF) ?

TABLE_PURCHASE_DETAIL

Customer ID	Store ID	Purchase Location
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles
3	2	New York
4	3	San Francisco

This table has a composite primary key [Customer ID, Store ID]. The non-key attribute is [Purchase Location].

In this case, [Purchase Location] only depends on [Store ID], which is only part of the primary key.

TABLE_PURCHASE

Customer ID	Store ID
1	1
1	3
2	1
3	2
4	3

TABLE_STORE

Store ID	Purchase Location
1	Los Angeles
2	New York
3	San Francisco

https://www.1keydata.com/database-normalization/second-normal-form-2nf.php

Third Normal Form (3NF)

A table is said to be in the Third Normal Form when

1. It is in the Second Normal form.
2. All the attributes in a table are determined only by the candidate keys of that relation and not by any non-prime attributes, i.e., *no Transitive Dependency*.
   - There is no transitive functional dependency

Third Normal Form (3NF) ?

id	name	city	country
1	Fuat	Ankara	Turkey
2	Zeynep	Columbus	USA
3	Dila	Zurich	Switzerland

Third Normal Form (3NF) ?

TABLE_BOOK_DETAIL

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

[Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type].

Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.

TABLE_BOOK

Book ID	Genre ID	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

https://www.1keydata.com/database-normalization/third-normal-form-3nf.php

Boyce-Codd Normal Form (BCNF)

- Main idea is that we define “good” and “bad” FDs as follows:
  - X → A is a *“good FD” if X is a (super)key*
    - In other words, if A is the set of all attributes
  - X → A is a *“bad FD”* otherwise
- We will try to eliminate the “bad” FDs!

Boyce-Codd Normal Form (BCNF)

- Why does this definition of “good” and “bad” FDs make sense?

- If X is *not* a (super)key, it functionally determines *some* of the attributes
  - *Recall*: this means there is redundancy
  - And redundancy like this can lead to data anomalies!

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation R is **in BCNF** if:

if $\{A_1, \ldots, A_n\} \rightarrow$ **B** is a *non-trivial* FD in R

then $\{A_1, \ldots, A_n\}$ **is a superkey** for R

Equivalently: ∀ sets of attributes X, either ($X^+ = X$) or (X^+ = all attributes)

In other words: there are no "bad" FDs

Boyce-Codd Normal Form

If all non-key attributes depend upon only the complete key, the relation is in BCNF.

Informally the Boyce-Codd normal form is expressed as "*Each attribute must represent a fact about the key, the whole key, and nothing but the key.*"

Example

Name	SSN	PhoneNumber	City
Fred	123-45-6789	206-555-1234	Seattle
Fred	123-45-6789	206-555-6543	Seattle
Joe	987-65-4321	908-555-2121	Westfield
Joe	987-65-4321	908-555-1234	Westfield

{SSN} → {Name,City}

This FD is *bad* because it is **not** a superkey

⟹ **Not** in BCNF

What is the key?
{SSN, PhoneNumber}

Example

Name	SSN	City
Fred	123-45-6789	Seattle
Joe	987-65-4321	Madison

SSN	PhoneNumber
123-45-6789	206-555-1234
123-45-6789	206-555-6543
987-65-4321	908-555-2121
987-65-4321	908-555-1234

{SSN} → {Name,City}

This FD is now *good* because it is the key

Let's check anomalies:

- Redundancy ?
- Update ?
- Delete ?

Now in BCNF!

Example

Author	Nationality	Book title	Genre	Number of pages
William Shakespeare	English	The Comedy of Errors	Comedy	100
Markus Winand	Austrian	SQL Performance Explained	Textbook	200
Jeffrey Ullman	American	A First Course in Database Systems	Textbook	500
Jennifer Widom	American	A First Course in Database Systems	Textbook	500

Author	Nationality
William Shakespeare	English
Markus Winand	Austrian
Jeffrey Ullman	American
Jennifer Widom	American

Book title	Genre	Number of pages
The Comedy of Errors	Comedy	100
SQL Performance Explained	Textbook	200
A First Course in Database Systems	Textbook	500

Author	Book title
William Shakespeare	The Comedy of Errors
Markus Winand	SQL Performance Explained
Jeffrey Ullman	A First Course in Database Systems
Jennifer Widom	A First Course in Database Systems

BCNF Decomposition Algorithm

BCNFDecomp(R):

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find *a set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

Find a set of attributes X which has non-trivial "bad" FDs, i.e. is not a superkey, using closures

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

<u>if</u> (not found) <u>then</u> **Return** R

If no "bad" FDs found, in BCNF!

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

<u>if</u> (not found) <u>**then**</u> **Return** R

<u>**let**</u> $Y = X^+ - X$, $Z = (X^+)^C$

Let Y be the attributes that ***X functionally determines*** (+ that are not in X)

And let Z be **the other attributes that it *doesn't***

BCNF Decomposition Algorithm

BCNFDecomp(R):

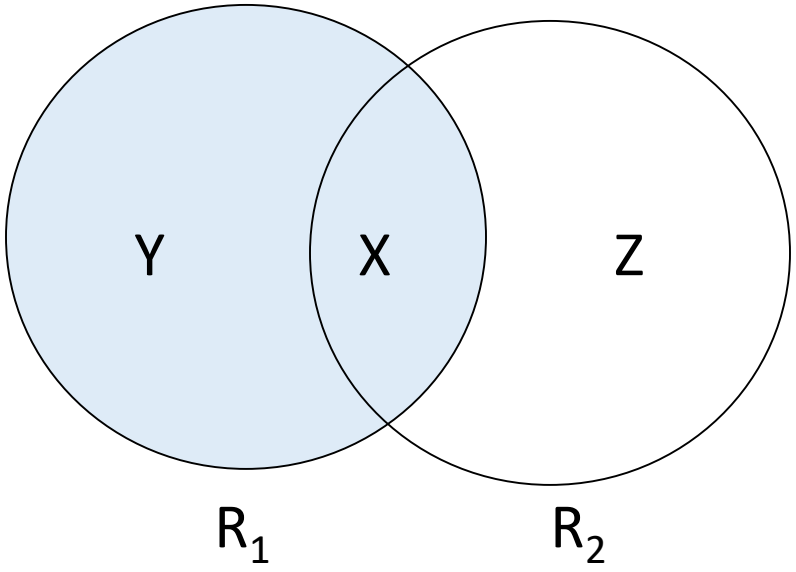
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

<u>if</u> (not found) <u>**then**</u> **Return** R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

Split into one relation (table) with X plus the attributes that X determines (Y)…



BCNF Decomposition Algorithm

BCNFDecomp(R):

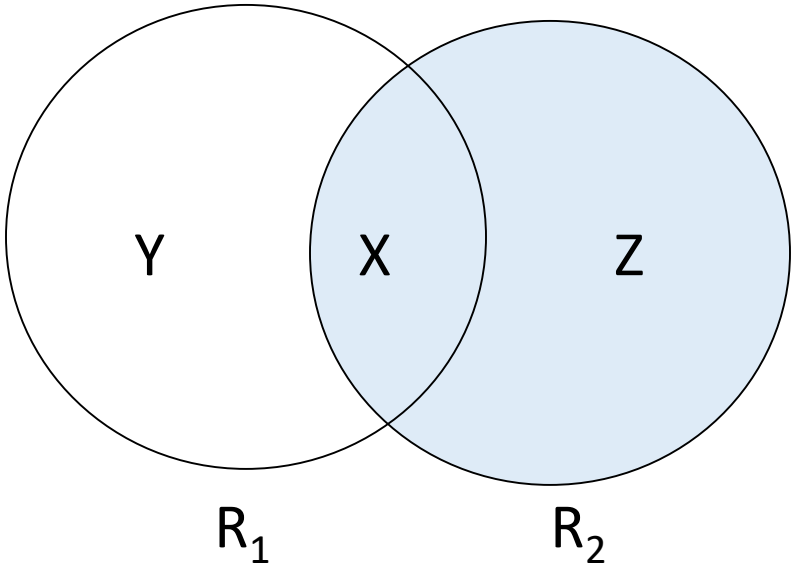
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

<u>if</u> (not found) <u>then</u> **Return** R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

And one relation with X plus the attributes it *does not* determine (Z)



BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

<u>if</u> (not found) <u>then</u> **Return** R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

Proceed recursively until no more "bad" FDs!

Example

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

<u>if</u> (not found) <u>then</u> **Return** R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

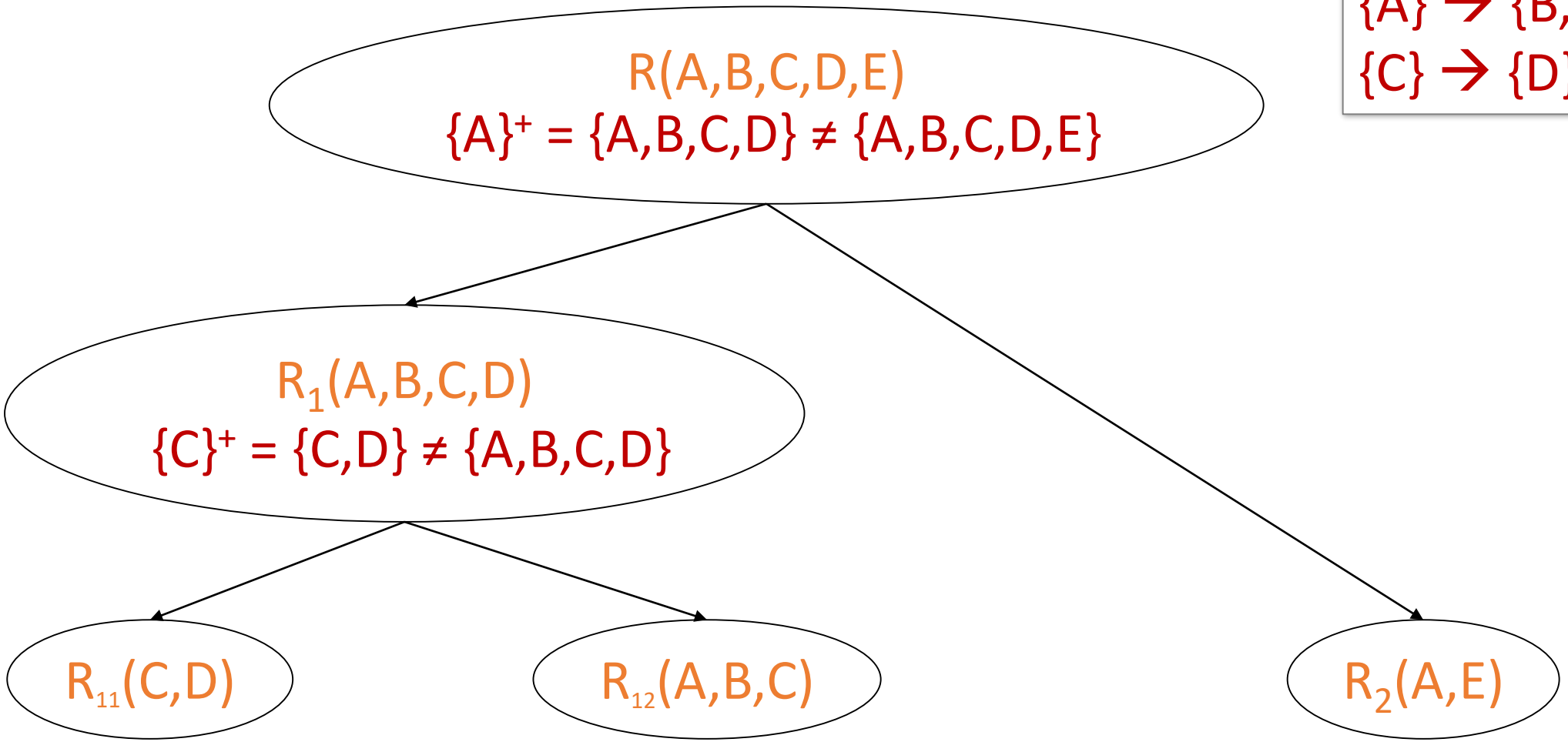
R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}

Example

R(A,B,C,D,E)

$\{A\} \rightarrow \{B,C\}$
$\{C\} \rightarrow \{D\}$



2. Decompositions

Recap: Decompose to remove redundancies

1. We saw that **redundancies** in the data ("bad FDs") can lead to data anomalies

2. We developed mechanisms to **detect and remove redundancies by decomposing tables into BCNF**
   1. BCNF decomposition is *standard practice*- very powerful & widely used!

3. However, sometimes decompositions can lead to **more subtle unwanted effects…**

When does this happen?

Decompositions in General

$R(A_1,\ldots,A_n,B_1,\ldots,B_m,C_1,\ldots,C_p)$

$R_1(A_1,\ldots,A_n,B_1,\ldots,B_m)$ $\quad$ $R_2(A_1,\ldots,A_n,C_1,\ldots,C_p)$

R_1 = the *projection* of R on $A_1, \ldots, A_n, B_1, \ldots, B_m$

R_2 = the *projection* of R on $A_1, \ldots, A_n, C_1, \ldots, C_p$

Theory of Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Price
Gizmo	19.99
OneClick	24.99
~~Gizmo~~	~~19.99~~

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Sometimes a decomposition is "correct"

i.e. it is a <u>Lossless decomposition</u>

Lossy Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Price	Category
19.99	Gadget
24.99	Camera
19.99	Camera

However sometimes it isn't

What's wrong here?

Lossless Decompositions

$R(A_1,\ldots,A_n,B_1,\ldots,B_m,C_1,\ldots,C_p)$

$R_1(A_1,\ldots,A_n,B_1,\ldots,B_m)$ $R_2(A_1,\ldots,A_n,C_1,\ldots,C_p)$

What (set) relationship holds between R1 Join R2 and R if lossless?

Hint: Which tuples of R will be present?

It's lossless if we have equality!

Lossless Decompositions

$R(A_1,\ldots,A_n,B_1,\ldots,B_m,C_1,\ldots,C_p)$

$R_1(A_1,\ldots,A_n,B_1,\ldots,B_m)$ $R_2(A_1,\ldots,A_n,C_1,\ldots,C_p)$

A decomposition R to (R1, R2) is **<u>lossless</u>** if R = R1 Join R2

Lossless Decompositions

$R(A_1,\ldots,A_n,B_1,\ldots,B_m,C_1,\ldots,C_p)$

$R_1(A_1,\ldots,A_n,B_1,\ldots,B_m)$ $\quad$ $R_2(A_1,\ldots,A_n,C_1,\ldots,C_p)$

If $\{A_1, \ldots, A_n\} \rightarrow \{B_1, \ldots, B_m\}$
Then the decomposition is lossless

Note: don't need
$\{A_1, \ldots, A_n\} \rightarrow \{C_1, \ldots, C_p\}$

BCNF decomposition is always lossless. Why?

A problem with BCNF

<u>Problem</u>: To enforce a FD, must reconstruct original relation—*on each insert!*

Note: This is historically inaccurate, but it makes it easier to explain

A Problem with BCNF

Unit	Company	Product
…	…	…

Unit	Company
…	…

Unit	Product
…	…

{Unit} → {Company}

{Unit} → {Company}
{Company,Product} → {Unit}

We do a BCNF decomposition on a "bad" FD:
$\{Unit\}^+ = \{Unit, Company\}$

We lose the FD {Company,Product} → {Unit}!!

So Why is that a Problem?

Unit	Company
Galaga99	UW
Bingo	UW

Unit	Product
Galaga99	Databases
Bingo	Databases

No problem so far. All *local* FD's are satisfied.

{Unit} → {Company}

Unit	Company	Product
Galaga99	UW	Databases
Bingo	UW	Databases

Let's put all the data back into a single table again:

Violates the FD {Company,Product} → {Unit}!!

The Problem

- We started with a table R and FDs F
- We decomposed R into BCNF tables R_1, R_2, … with their own FDs F_1, F_2, …
- We insert some tuples into each of the relations—which satisfy their local FDs but when reconstruct it violates some FD **across** tables!

<u>Practical Problem</u>: To enforce FD, must reconstruct R—*on each insert!*

Possible Solutions

- Various ways to handle so that decompositions are all lossless / no FDs lost
  - For example 3NF- stop short of full BCNF decompositions.

- Usually a tradeoff between redundancy / data anomalies and FD preservation…

BCNF still most common- with additional steps to keep track of lost FDs…

3. MVDs

What you will learn about in this section

1. MVDs

Multi-Value Dependencies (MVDs)

- A multi-value dependency (MVD) is another type of dependency that could hold in our data, ***which is not captured by FDs***

- Formal definition:
  - Given a relation **R** having attribute set **A**, and two sets of attributes $\mathbf{X}, \boldsymbol{Y} \subseteq \boldsymbol{A}$
  - The ***multi-value dependency (MVD)*** $\boldsymbol{X} \twoheadrightarrow \boldsymbol{Y}$ holds on R if
  - ***for any tuples*** $\boldsymbol{t_1}, \boldsymbol{t_2} \in \boldsymbol{R}$ s.t. $\boldsymbol{t_1}[\boldsymbol{X}] = \boldsymbol{t_2}[\boldsymbol{X}]$***,*** there exists a tuple $\boldsymbol{t_3}$ s.t.:
    - **t_1[X] = t_2[X] = t_3[X]**
    - **t_1[Y] = t_3[Y]**
    - **t_2[A\Y] = t_3[A\Y]**
      - *Where A \ B means "elements of set A not in set B"*

Multi-Value Dependencies (MVDs)

- One less formal, literal way to phrase the definition of an MVD:

- ***The MVD*** $\boldsymbol{X \twoheadrightarrow Y}$ holds on R if for any pair of tuples with the same X values, the "swapped" pair of tuples with the same X values, but the other permutations of Y and A\Y values, is also in R

Ex: X = {x}, Y = {y}:

x	y	z
1	0	1
1	1	0

For $\boldsymbol{X \twoheadrightarrow Y}$ to hold must have…

x	y	z
1	0	1
1	1	0
1	0	0
1	1	1

Note the connection to a local *cross-product…*

Multi-Value Dependencies (MVDs)

- Another way to understand MVDs, in terms of *conditional independence:*

- ***The MVD*** $\boldsymbol{X \twoheadrightarrow Y}$ holds on R if given X, Y is conditionally independent of A \ Y and vice versa…

Here, given x = 1, we know for ex. that:
y = 0 → z = 1

I.e. z is conditionally ***dependent*** on y given x

x	y	z
1	0	1
1	1	0

Here, this is not the case!

I.e. z is conditionally ***independent*** of y given x

x	y	z
1	0	1
1	1	0
1	0	0
1	1	1

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

Are there any functional dependencies that might hold here?

No…

And yet it seems like there is some pattern / dependency…

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre…

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre…

Given a set of movies and snacks…

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre…

Given a set of movies and snacks…

Any movie / snack combination is possible!

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write **{A} ↠ {B}** if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write **{A} ↠ {B}** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write **{A} ↠ {B}** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write **{A} ↠ {B}** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[R\backslash B] = t_2[R\backslash B]$

Where R\B is "R minus B" i.e. the attributes of R not in B

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_2	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
t_3	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_1	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

Note this also works!

Remember, an MVD holds over *a relation or an instance*, so defn. must hold for every applicable pair…

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_2	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
t_3	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_1	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

This expresses a sort of dependency (= data redundancy) that we *can't* express with FDs

**Actually, it expresses* *<u>conditional independence</u>* *(between film and snack given movie theatre)!*

Example

Students

Student_Name	Major	Sport
Ravi	Art History	Soccer
Ravi	Art History	Volleyball
Ravi	Art History	Tennis
Beth	Chemistry	Tennis
Beth	Chemistry	Soccer

This is read as "Student_Name multidetermines Major" and "Student_Name multidetermines Sport."

Students & Majors

Student_Name	Major
Ravi	Art History
Ravi	Art History
Ravi	Art History
Beth	Chemistry
Beth	Chemistry

Students & Sports

Student_Name	Sport
Ravi	Soccer
Ravi	Volleyball
Ravi	Tennis
Beth	Tennis
Beth	Soccer

Summary

- Normal forms describe how to **remove** this redundancy by **decomposing** relations
- Normalization let us avoid anomalies
- Normalization saves space, but space is cheap
- Normalization simplifies updates, but read are more common
- Normalization does not help in optimizing queries much

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the *CS145 Introduction to Databases* lecture given by *Christopher Ré* at *Stanford University* (http://web.stanford.edu/class/cs145/).

3NF
Example

Ques 1. Consider a relation A with attributes A(P,Q,R,S,T,U,V,W) and given FDs:

P→RS
PRU→V
PS→QTU
QRV→S
RU→PW
RW→U
S→Q
W→STV

1) Compute all keys for A.
2) Compute a minimal cover.
3) Using minimal cover, apply 3NF synthesis algorithm to obtain a lossless and dependency-preserving decomposition of relation A.

Solution

1) Compute all keys for A.

- Examining all subsets of the attributes would be very time-consuming because there are 2^8 of them.
- By inspection, we can see that P^+ = PQRSTUVW, which means that P is a key and no superset of P can be a key.

- Also, RU+ = RUPWSTVQ, which means that RU is a key and no superset of RU can be a key. (That means that R alone or U alone could be part of a key, but RU cannot.)
- But there is no key that has R but not U. We can determine this because even if we use every other attribute except P (which we know can't be in any other key), we don't have a key: $QSTRVW^+$ = QSTRVW
- Similarly, there is no key that has U but not R. We know this because even if we use every other attribute except P (which we know can't be in any other key), we don't have a key: $QSTUVW^+$ = QSTUVW
- Therefore, the only keys are P and RU.

2) Minimal cover

Step1: Single attribute @ RHS

1. P→R
2. P→S
3. PRU→V
4. PS→Q
5. PS→T
6. PS→U
7. QRV→S
8. RU→P
9. RU→W
10. RW→U
11. S→Q
12. W→S
13. W→T
14. W→V

Step 2: Reduce LHS

			Final FDs (after reduction)
1	P→R	Cannot be reduced	P→R
2	P→S	Cannot be reduced	P→S
3	PRU→V	Can be reduced as P→R	PU→V
4	PS→Q	Can be reduced as P→S	P→Q
5	PS→T	Can be reduced as P→S	P→T
6	PS→U	Can be reduced as P→S	P→U
7	QRV→S	Cannot be reduced	QRV→S
8	RU→P	Cannot be reduced	RU→P
9	RU→W	Cannot be reduced	RU→W
10	RW→U	Cannot be reduced	RW→U
11	S→Q	Cannot be reduced	S→Q
12	W→S	Cannot be reduced	W→S
13	W→T	Cannot be reduced	W→T
14	W→V	Cannot be reduced	W→V

Step 3: Eliminate redundant FDs

		Closure (considering FD)	**Closure without considering FD**	**Result (Keep/ Discard)**	**Final set**
1	P→R	P^+= PRSQTUW	P^+= PSQTU There is no way to get R without this FD	Keep	P→R
2	P→S	P^+= PSRUWQTV	P^+= PRQUWSTV	Discard	
3	~~PU→V~~	PU^+ = PUVRSQTW	PU^+ = PURSQTWV	Discard	
4	~~P→Q~~	P^+ = PSQRTUW	P^+ = PRQTUW	Discard	
5	~~P→T~~	P^+ = PSQRTUW	P^+ = PSRUWQT	Discard	
6	P→U	P^+ = PSQRTUW	P^+ =PSRQ	Keep	P→U
7	QRV→S	QRV^+= QRVS	QRV^+= QRV	Keep	QRV→S
8	RU→P	RU^+=RUPSWQT	RU^+= RUWSTQ There is no way to get P without this FD	Keep	RU→P
9	RU→W	RU^+=RUPSWQT	RU^+=RUPSQ	Keep	RU→W
10	~~RW→U~~	RW^+ = RWUSTPQ	RW^+ = RWSTUPQ	Discard	
11	S→Q	S^+ = SQ	S^+ = S	Keep	S→Q
12	W→S	W^+ = WSTVQ	W^+ = WTV	Keep	W→S
13	W→T	W^+ = WSTVQ	W^+ = WSVQ	Keep	W→T
14	W→V	W^+ = WVSTQ	W^+ = WTSQ	Keep	W→V

Minimal cover

1. P→R
2. P→U
3. QRV→S
4. RU→P
5. RU→W
6. S→Q
7. W→S
8. W→T
9. W→V

3) Decomposition using 3NF synthesis

Following the 3NF synthesis algorithm, we would get one relation for each FD. However, we can merge the right-hand sides before doing so. This will yield a smaller set of relations and they will still form a lossless and dependency-preserving decomposition of relation R into a collection of relations that are in 3NF. So the final minimal set will be as follows:

1. P→RU
2. QRV→S
3. RU→PW
4. S→Q
5. W→STV

The set of relations that would result would have these attributes:

R1(PRU);
R2(QRVS);
R3(RUPW);
R4(SQ);
R5(WSTV)

Since the attributes, SQ occur within R2, we don't need to keep the relation R4. Similarly, since the attributes PRU occur in R3, we don't need to keep the relation R1. So, the resultant relations are:

R1(QRVS);
R2(RUPW);
R3(WSTV)

P is a key of R, so there is no need to add another relation that includes a key.

Therefore, the final set of relations is:
R1(QRVS);
R2(RUPW);
R3(WSTV)

Transactions

BBM471 Database Management Systems

Goals for this lecture

- **Transactions** are a programming abstraction that enables the DBMS to handle *recovery* and *concurrency* for users.

- **Application:** Transactions are critical for users
  - Even casual users of data processing systems!

- **Fundamentals:** The basics of **how** TXNs work
  - Transaction processing is part of the debate around new data processing systems

  - Give you enough information to understand how TXNs work, and the main concerns with using them

Today's Lecture

1. Transactions
2. Properties of Transactions: ACID
3. Logging
4. Concurrency, scheduling & anomalies
5. Locking: 2PL, conflict serializability, deadlock detection

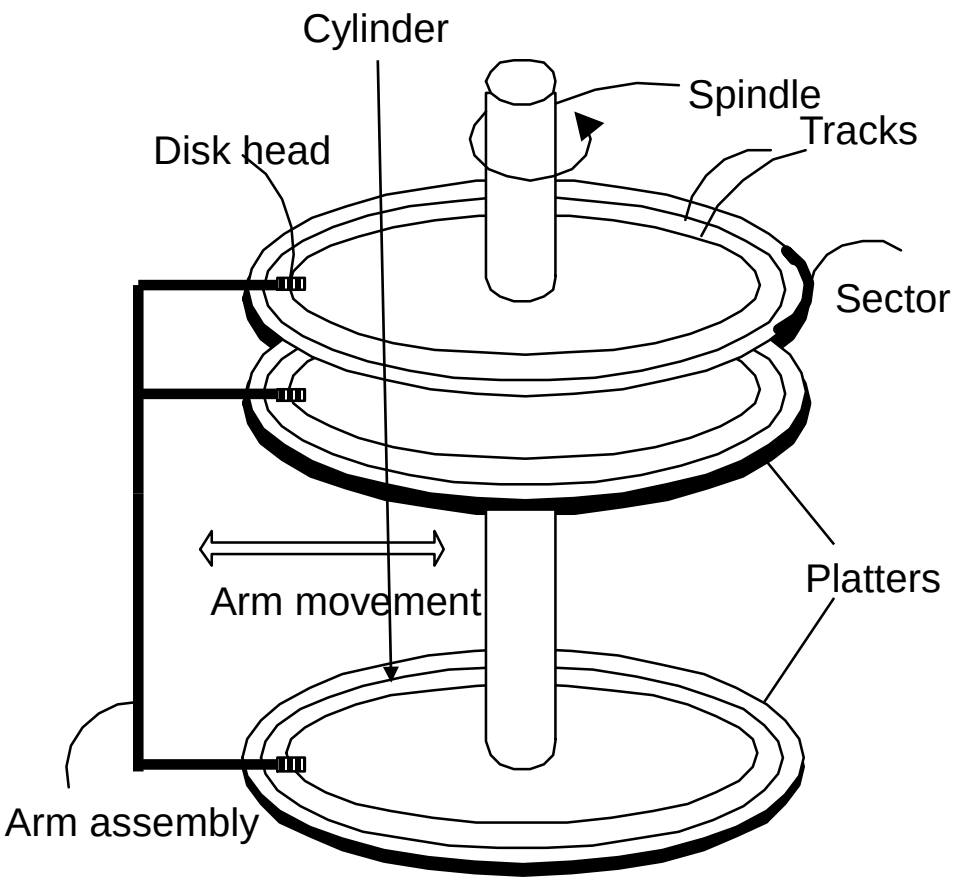
1. Transactions

What you will learn about in this section

1. Our “model” of the DBMS / computer
2. Transactions basics
3. Motivation: Recovery & Durability
4. Motivation: Concurrency

High-level: Disk vs. Main Memory

- **Disk**:
  - *Slow*
    - Sequential access
      - (although fast sequential reads)
  - *Durable*
    - We will assume that once on disk, data is safe!
  - Cheap



Forget about SSDs for a while :-)

High-level: Disk vs. Main Memory

- Random Access Memory (RAM) or **Main Memory**:
  - *Fast*
    - Random access, byte addressable
      - ~10x faster for <u>sequential access</u>
      - ~100,000x faster for <u>random access!</u>
  - *Volatile*
    - Data can be lost if e.g. crash occurs, power goes out, etc!
  - Expensive
    - For $100, get 16GB of RAM vs. 2TB of disk!

Our model: Three Types of Regions of Memory

	Local	Global
Main Memory (RAM)	1	2 *4*
Disk		3

1. **Local:** In our model each process in a DBMS has its own local memory, where it stores values that only it "sees"

2. **Global:** Each process can read from / write to shared data in main memory

3. **Disk:** Global memory can read from / flush to disk

4. ***Log:*** *Assume on stable disk storage- spans both main memory and disk…*

Log is a *sequence* from main memory -> disk

"Flushing to disk" = writing to disk from main memory

High-level: Disk vs. Main Memory

- Keep in mind the tradeoffs here as motivation for the mechanisms we introduce
  - Main memory: fast but limited capacity, volatile
  - vs. Disk: slow but large capacity, durable

How do we effectively utilize ***both*** ensuring certain critical guarantees?

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = 'Gizmo'
COMMIT
```

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

Examples:

- Transfer money between accounts
- Purchase a group of products
- Register for a class (either waitlist or allocated)

Transactions in SQL

- In “ad-hoc” SQL:
  - Default: each statement = one transaction

- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 100
        WHERE name = ‘Bob’;
        UPDATE Bank SET amount = amount + 100
        WHERE name = ‘Joe’;
COMMIT
```

Model of Transaction for BBM471

Note: For BBM471, we assume that the DBMS *only* sees <u>reads and writes to data</u>

- User may do much more
- In real systems, databases do have more info...

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. **<u>Recovery & Durability</u>**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

2. **<u>Concurrency:</u>** Achieving better performance by parallelizing TXNs *without* creating anomalies

Motivation

<u>1. Recovery & Durability</u> of user data is essential for reliable DBMS usage

- The DBMS may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either **durably stored in full**, **or not at all**; keep log to be able to "roll-back" TXNs

Protection against crashes / aborts

Client 1:

```
INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99
```

Crash / abort!

```
DELETE Product
        WHERE price <=0.99
```

What goes wrong?

Protection against crashes / aborts

```
Client 1:
        START TRANSACTION
                INSERT INTO SmallProduct(name, price)
                        SELECT pname, price
                        FROM Product
                        WHERE price <= 0.99

                DELETE Product
                        WHERE price <=0.99
        COMMIT OR ROLLBACK
```

Now we'd be fine! We'll see how / why this lecture

Motivation

2. Concurrent execution of user programs is essential for good DBMS performance.

- Disk accesses may be frequent and **slow**- optimize for throughput (# of TXNs), trade for latency (time for any one TXN)
- Users should still be able to execute TXNs as if in **isolation** and such that **consistency** is maintained

Idea: Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs humming…

Multiple users: single statements

```
Client 1: UPDATE Product
                SET Price = Price – 1.99
                WHERE pname = ‘Gizmo’

Client 2:       UPDATE Product
                SET Price = Price * 0.5
                WHERE pname=‘Gizmo’
```

Two managers attempt to discount products *concurrently*-
What could go wrong?

Multiple users: single statements

```
Client 1: START TRANSACTION
                    UPDATE Product
                    SET Price = Price – 1.99
                    WHERE pname = ‘Gizmo’
          COMMIT

Client 2: START TRANSACTION
                    UPDATE Product
                    SET Price = Price * 0.5
                    WHERE pname=‘Gizmo’
          COMMIT
```

Now works like a charm- we’ll see how / why next lecture…

2. Properties of Transactions

What you will learn about in this section

1. Atomicity
2. Consistency
3. Isolation
4. Durability

Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (i.e. looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

ACID: Atomicity

- TXN's activities are atomic: **all or nothing**
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*
- Two possible outcomes for a TXN
  - It ***commits***: all the changes are made
  - It ***aborts***: no changes are made

ACID: Consistency

- The tables must always satisfy user-specified ***integrity constraints***
  - *Examples:*
    - Account number is unique
    - Stock amount can't be negative
    - Sum of *debits* and of *credits* is 0

- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - *System* makes sure that the txn is **atomic**

ACID: Isolation

- A transaction executes concurrently with other transactions

- **Isolation**: the effect is as if each transaction executes in *isolation* of the others.
  - Should not be able to observe changes from other transactions during the run

ACID: Durability

- The effect of a TXN must continue to exist (***"persist"***) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.

- Means: Write data to **disk**

Change on the horizon?
Non-Volatile Ram (NVRam).
Byte addressable.

Challenges for ACID properties

- In spite of failures: Power failures, but not media failures
- Users may abort the program: need to “rollback the changes”
  - Need to *log* what happened
- Many users executing concurrently
  - Can be solved via locking

And all this with… Performance!!

A Note: ACID is contentious!

- Many debates over ACID, both **historically** and **currently**

- Many newer "NoSQL" DBMSs relax ACID
  - NoSQL means NOT Only SQL

- In turn, now "NewSQL" reintroduces ACID compliance to NoSQL-style DBMSs…

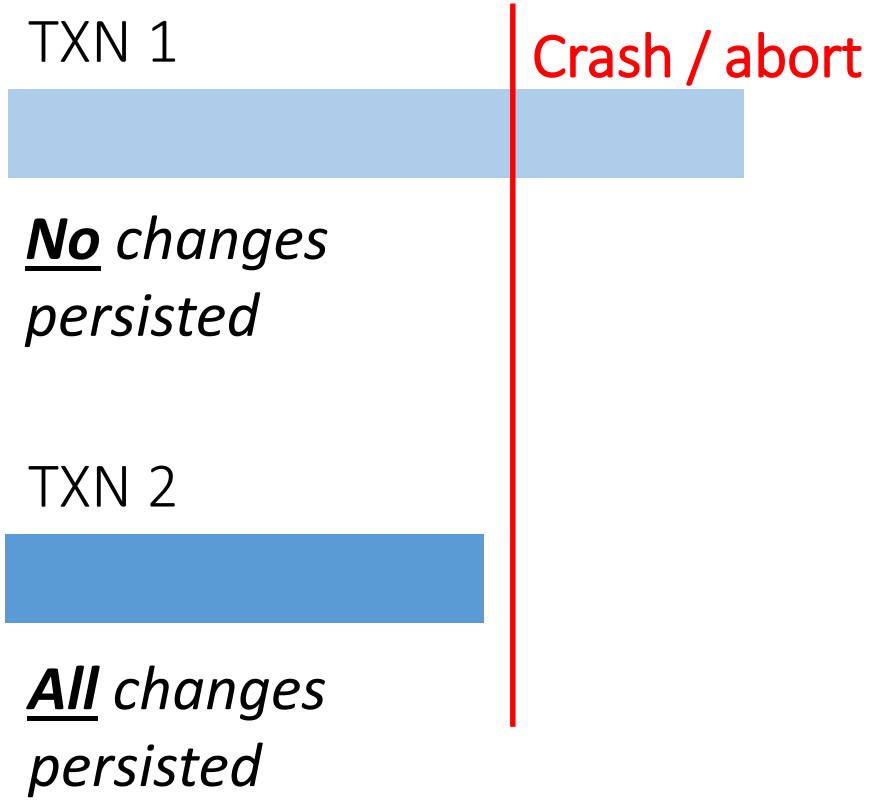
- BASE
  - Basically Available, Soft State, Eventual Consistency

Goal for this lecture: Ensuring Atomicity & Durability

ACID

- **Atomicity:**
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen

- **Durability:**
  - If DBMS stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*



We'll focus on how to accomplish atomicity (via logging)

The Log

- Is a list of modifications
- Log is *duplexed* and *archived* on stable storage.
- Can **force write** entries to disk
  - A page goes to disk.
- All log activities ***handled transparently*** by the DBMS.

Assume we don't lose it!

Basic Idea: (Physical) Logging

- Record UNDO information for every update!
  - Sequential writes to log
  - Minimal info (diff) written to log

- The **log** consists of **an ordered list of actions**
  - Log record contains:

    <XID, location, old data, new data>

This is sufficient to UNDO any transaction!

Why do we need logging for atomicity?

- Couldn't we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and time, this could work…*

- However, we **need to log partial results of TXNs** because of:
  - Memory constraints (enough space for full TXN??)
  - Time constraints (what if one TXN takes very long?)

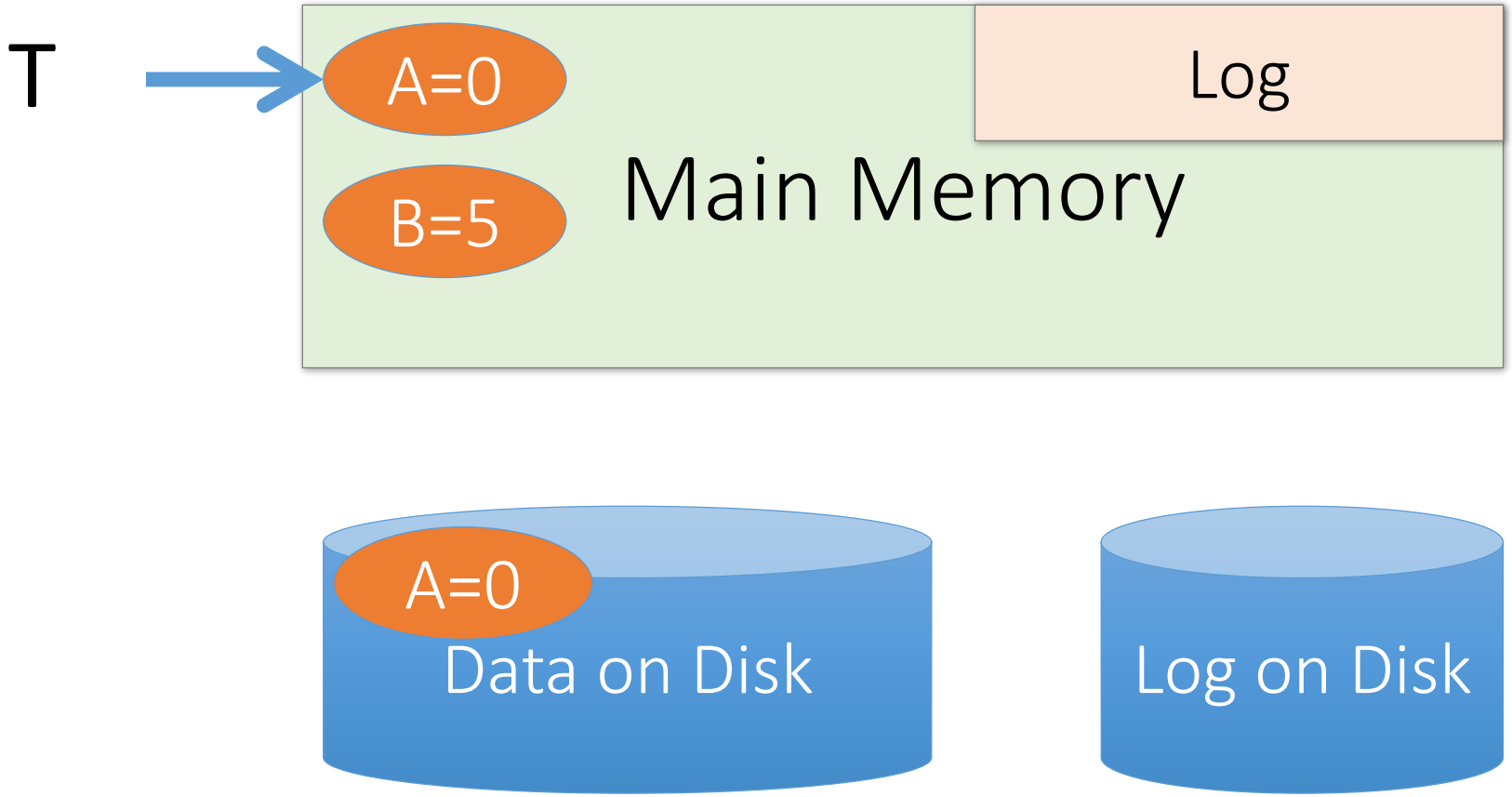
We need to write partial results to disk!
…And so we need a **log** to be able to ***undo*** these partial results!

3. Atomicity & Durability via Logging

A picture of logging

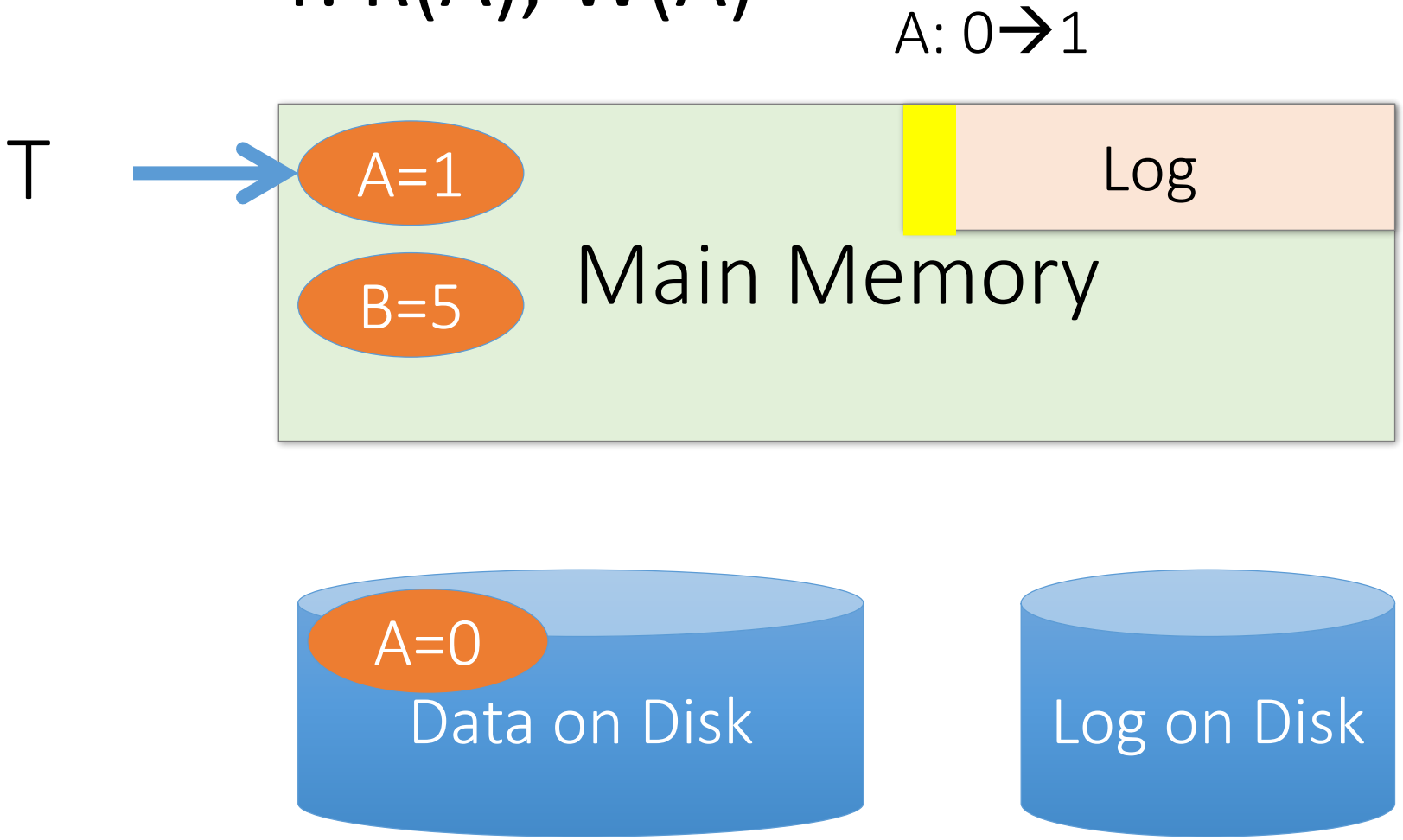
T: R(A), W(A)

T Transaction
R Read Operation
W Write Operation
A Data Item



A picture of logging

T: R(A), W(A)



A picture of logging

T: R(A), W(A)

A: 0→1

Operation recorded in log in main memory!

What is the correct way to write this all to disk?

- We'll look at the *Write-Ahead Logging (WAL)* protocol

- We'll see why it works by looking at other protocols which are incorrect!

Remember: Key idea is to ensure durability *while* maintaining our ability to "undo"!

Write-Ahead Logging (WAL)
TXN Commit Protocol

Transaction Commit Process

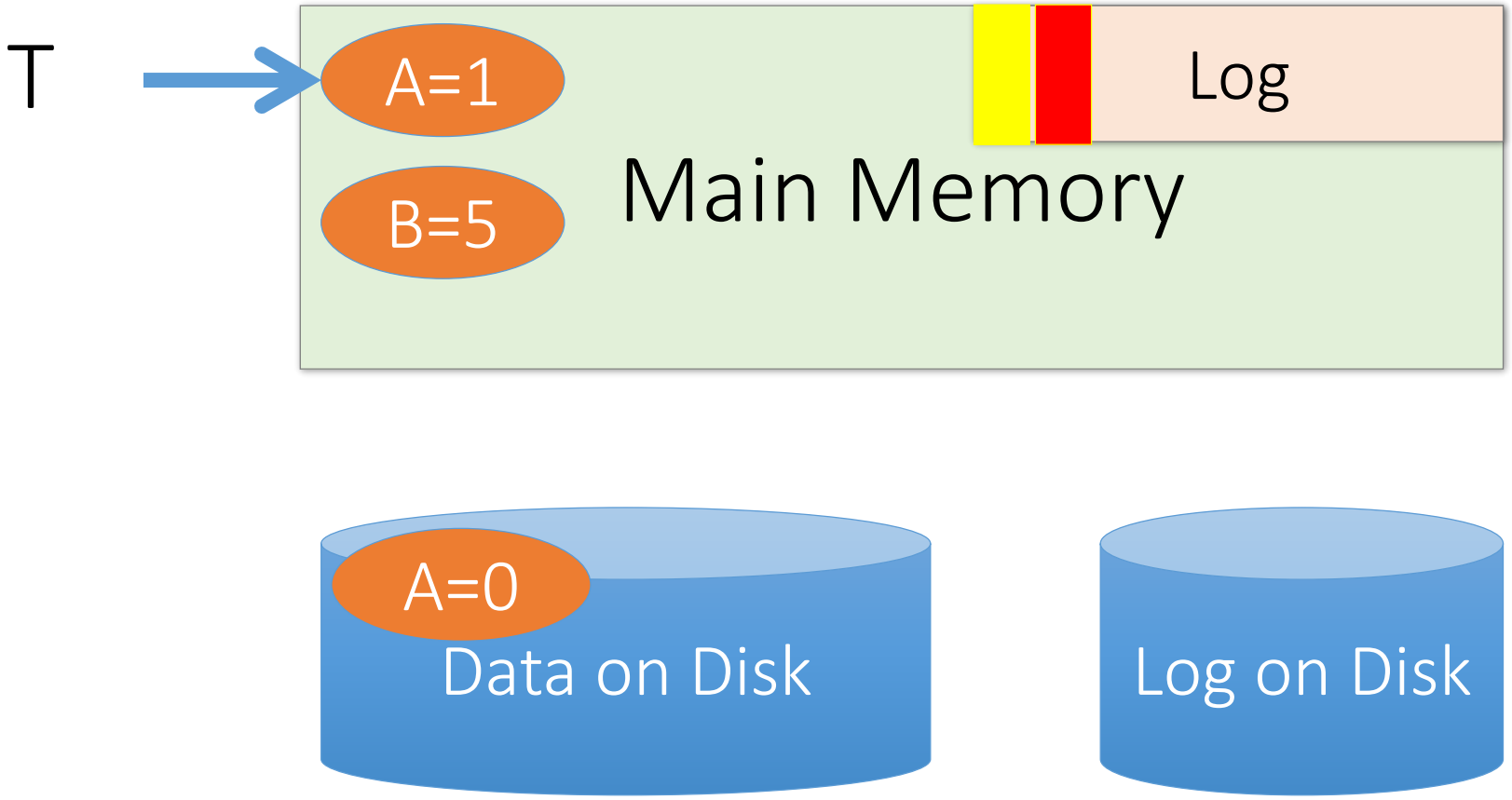
1. FORCE Write **commit** record to log
2. All log records up to last update from this TX are FORCED
3. Commit() returns

Transaction is committed *once commit log record is on stable storage*

Incorrect Commit Protocol #1

T: R(A), W(A)

A: 0→1



Let's try committing *before* we've written either data or log to disk…

OK, Commit!

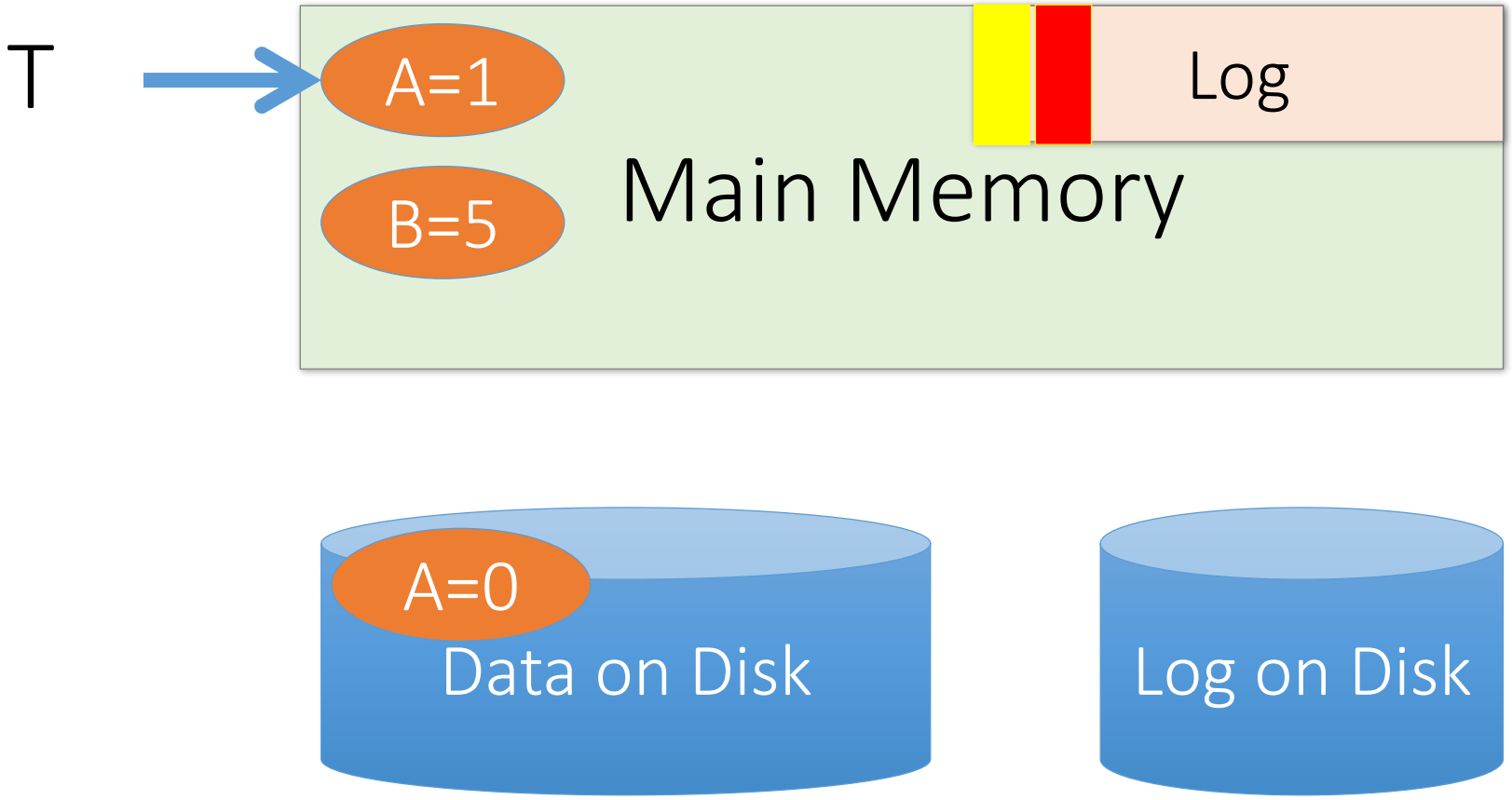
If we crash now, is T durable?

Lost T's update!

Incorrect Commit Protocol #2

T: R(A), W(A)

A: 0→1



Let's try committing *after* we've written data but *before* we've written log to disk…

OK, Commit!

If we crash now, is T durable? Yes! Except…

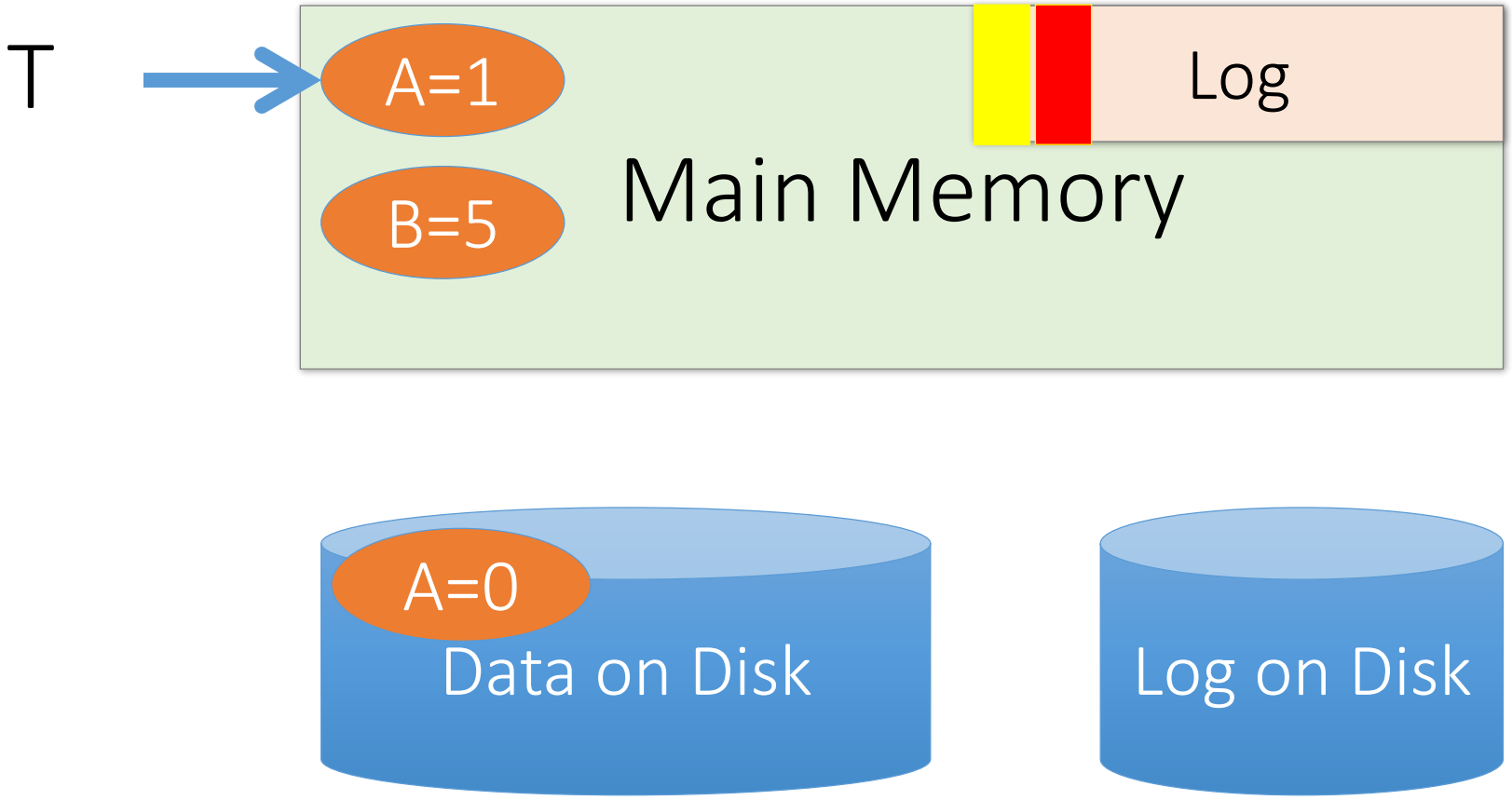
How do we know whether T was committed??

Improved Commit Protocol (WAL)

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

A: 0→1



This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**... this is WAL!

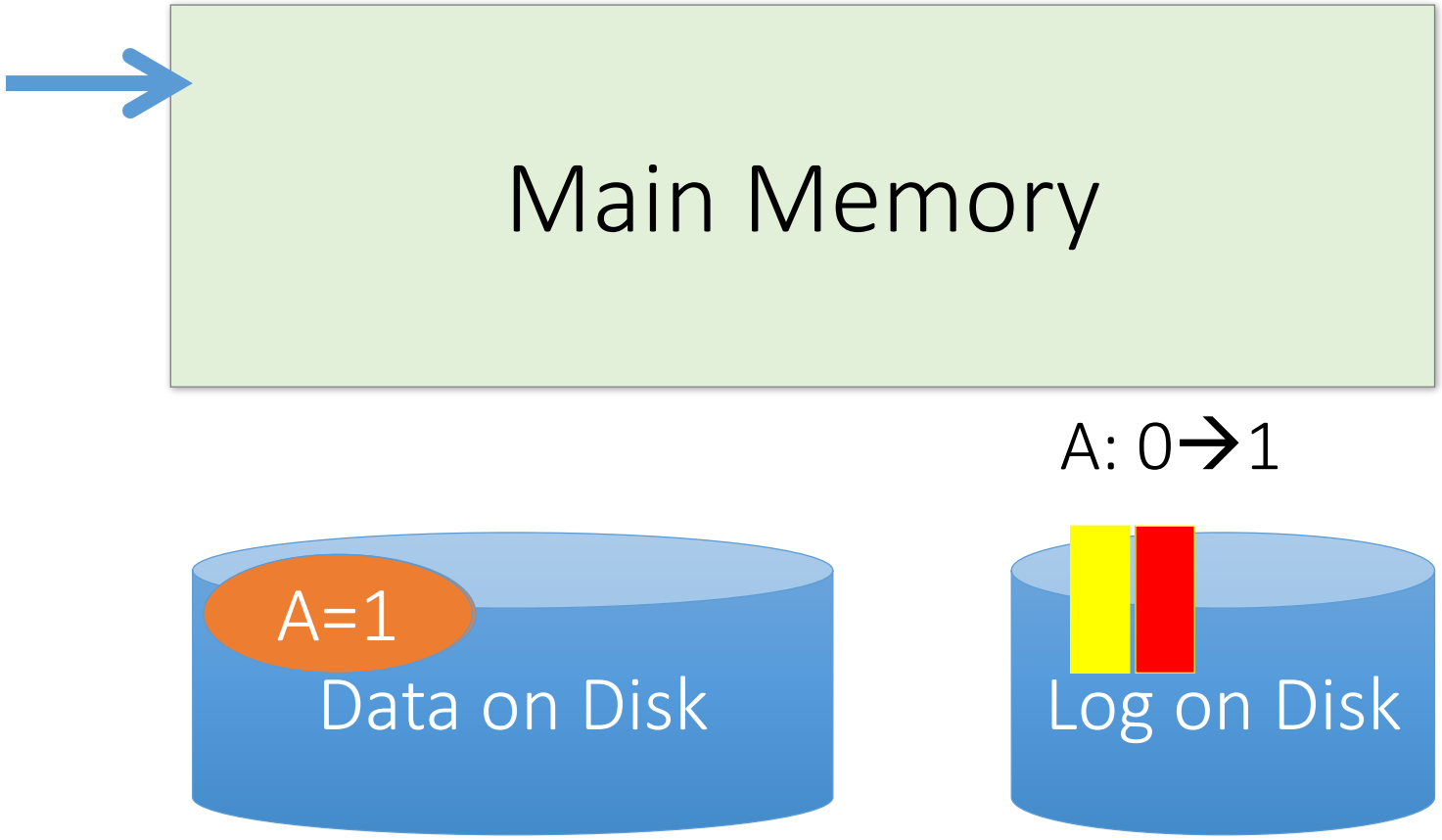
OK, Commit!

If we crash now, is T durable?

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

T



This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**... this is WAL!

OK, Commit!

If we crash now, is T durable?

USE THE LOG!

Write-Ahead Logging (WAL)

- DB uses **Write-Ahead Logging (WAL)** Protocol:

Each update is logged! Why not reads?

1. Must *force log record* for an update *before* the corresponding data page goes to storage

→ <u>Atomicity</u>

2. Must *write all log records* for a TX *before commit*

→ <u>Durability</u>

Logging Summary

- If DB says TX **commits**, TX effect **remains** after database crash
- DB can **undo actions** and help us with **atomicity**
- This is only half the story…

4. Concurrency, Scheduling & Anomalies

What you will learn about in this section

1. Interleaving & scheduling

2. Conflict & anomaly types

Concurrency: Isolation & Consistency

- The DBMS must handle concurrency such that…

1. **Isolation** is maintained: Users must be able to execute each TXN **as if they were the only user**
   - DBMS handles the details of *interleaving* various TXNs

ACID

2. **Consistency** is maintained: TXNs must leave the DB in a **consistent state**
   - DBMS handles the details of enforcing integrity constraints

ACID

Example- consider two TXNs:

```
T1: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt + 100
        WHERE Name = 'A'

        UPDATE Accounts
        SET Amt = Amt - 100
        WHERE Name = 'B'
COMMIT
```

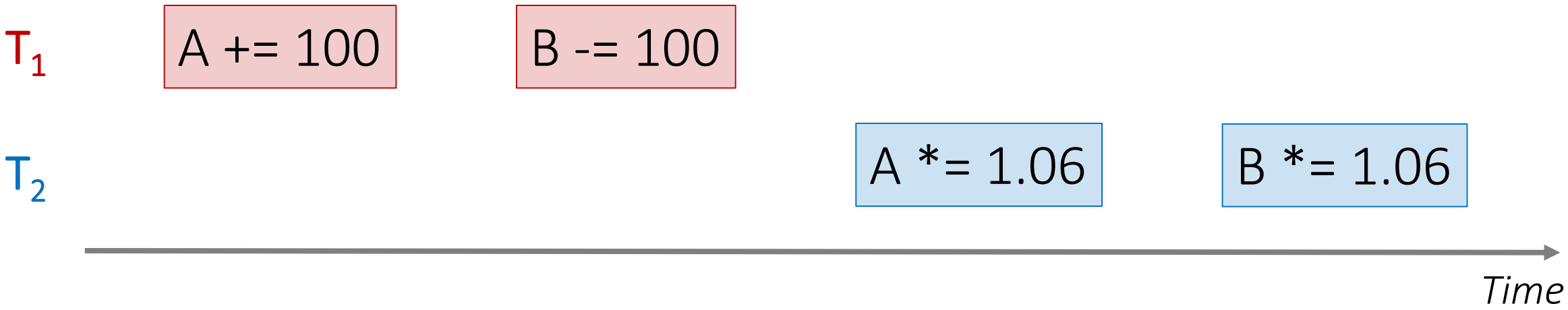
```
T2: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt * 1.06
COMMIT
```

T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

We can look at the TXNs in a timeline view- serial execution:



T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

The TXNs could occur in either order… DBMS allows!

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

Time

T2 credits both accounts with a 6% interest payment

T1 transfers $100 from B's account to A's account

Example- consider two TXNs:

The DBMS can also **interleave** the TXNs

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

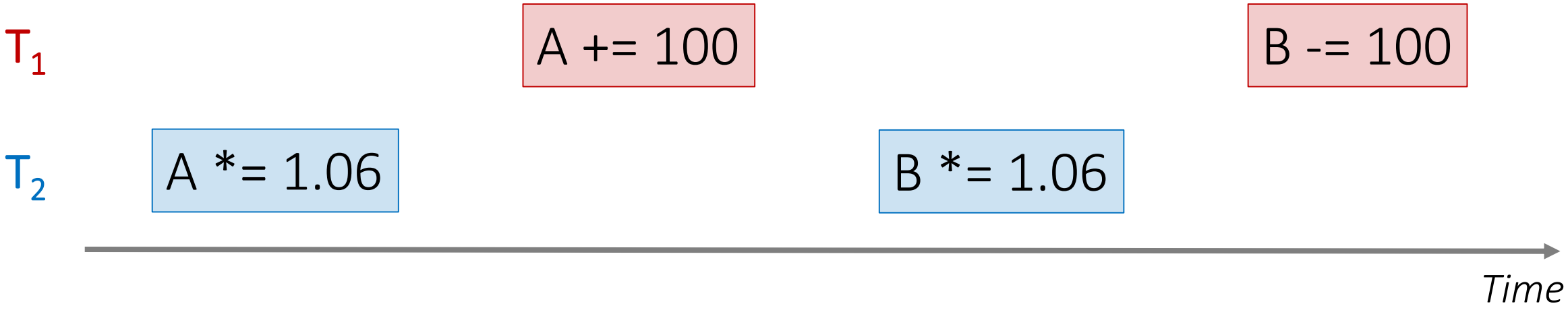
Time

T2 credits A's account with 6% interest payment, then T1 transfers $100 to A's account…

T2 credits B's account with a 6% interest payment, then T1 transfers $100 from B's account…

Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



What goes wrong here??

Recall: Three Types of Regions of Memory

	Local	Global
Main Memory (RAM)	1	2 *4*
Disk		3

1. **Local:** In our model each process in a DBMS has its own local memory, where it stores values that only it "sees"

2. **Global:** Each process can read from / write to shared data in main memory

3. **Disk:** Global memory can read from / flush to disk

4. ***Log:*** *Assume on stable disk storage- spans both main memory and disk…*

Log is a *sequence* from main memory -> disk

"Flushing to disk" = writing to disk.

Why Interleave TXNs?

- Interleaving TXNs might lead to anomalous outcomes… why do it?

- Several important reasons:
  - Individual TXNs might be *slow*- don't want to block other users during!

  - Disk access may be *slow*- let some TXNs use CPUs while others accessing disk!

All concern large differences in ***performance***

Interleaving & Isolation

- The DBMS has freedom to interleave TXNs
- However, it must pick an interleaving or **schedule** such that isolation and consistency are maintained
  - Must be *as if* the TXNs had executed serially!

"With great power comes great responsibility"

ACID

DBMS must pick a schedule which maintains isolation & consistency

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule T_1,T_2:

T_1: A += 100 → B -= 100

T_2: A *= 1.06 → B *= 1.06

A	B
$159	$106

Interleaved schedule A:

T_1: A += 100 → B -= 100

T_2: A *= 1.06 → B *= 1.06

A	B
$159	$106

Same result!

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule T_1,T_2:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$159	$106

Interleaved schedule B:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$159	$112

Different result than serial T_1,T_2!

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule T_2,T_1:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$153	$112

Interleaved schedule B:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$159	$112

Different result than serial T_2,T_1 ALSO!

Scheduling examples

Interleaved schedule B:

T_1	A += 100			B -= 100
T_2		A *= 1.06	B *= 1.06	

This schedule is different than ***any serial order!*** We say that it is **not serializable**

Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions

- A and B are **equivalent schedules** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B

- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some**" makes this definition powerful & tricky!

Serializable?

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

Serial schedules:

	A	B
T_1,T_2	1.06*(A+100)	1.06*(B-100)
T_2,T_1	1.06*A + 100	1.06*B - 100

A	B
1.06*(A+100)	1.06*(B-100)

Same as a serial schedule *for all possible values of A, B* = <u>serializable</u>

Serializable?

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

Serial schedules:

	A	B
T_1,T_2	1.06*(A+100)	1.06*(B-100)
T_2,T_1	1.06*A + 100	1.06*B - 100

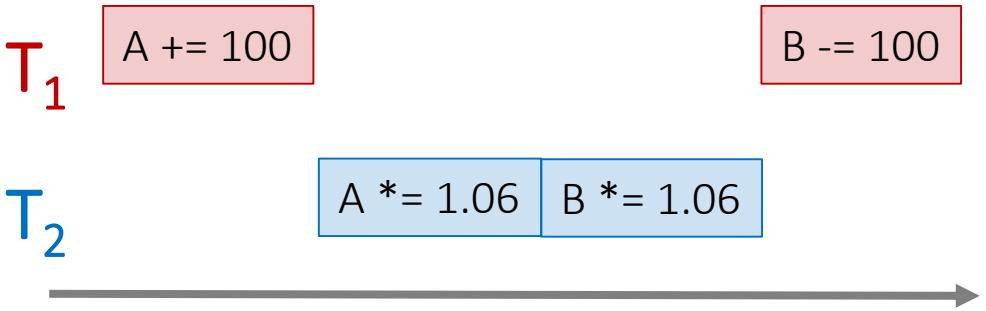
A	B
1.06*(A+100)	1.06*B - 100

Not *equivalent* to any serializable schedule = ***not*** **serializable**

What else can go wrong with interleaving?

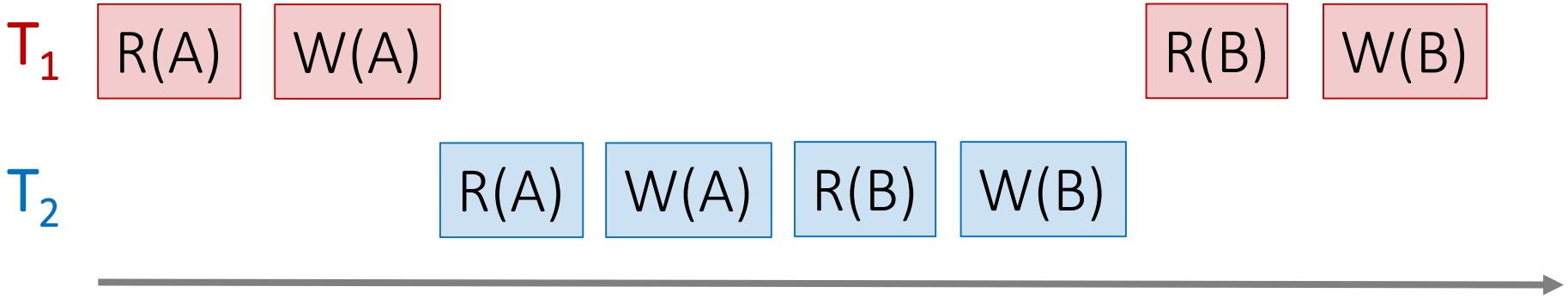
- Various anomalies which break isolation / serializability
  - Often referred to by name…
- Occur because of / with certain “conflicts” between interleaved TXNs

The DBMS's view of the schedule



Each action in the TXNs *reads a value from global memory* and then *writes one back to it*

Scheduling order matters!



Conflict Types

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write

- Thus, there are three types of conflicts:
  - Read-Write conflicts (RW)
  - Write-Read conflicts (WR)
  - Write-Write conflicts (WW)

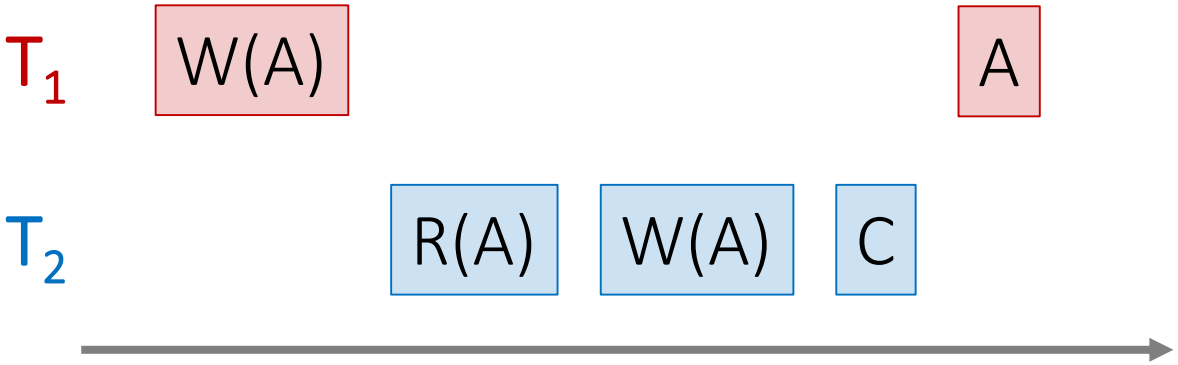
Why no "RR Conflict"?

Interleaving anomalies occur with / because of these conflicts between TXNs *(but these conflicts can occur without causing anomalies!)*

Classic Anomalies with Interleaved Execution

"Dirty read" / Reading uncommitted data:

Example:



1. T_1 <u>writes</u> some data to A
2. T_2 <u>reads</u> from A, then writes back to A & commits
3. T_1 then aborts- *now T_2's result is based on an obsolete / inconsistent value*

*Occurring with / because of a **WR conflict***

Dirty Read

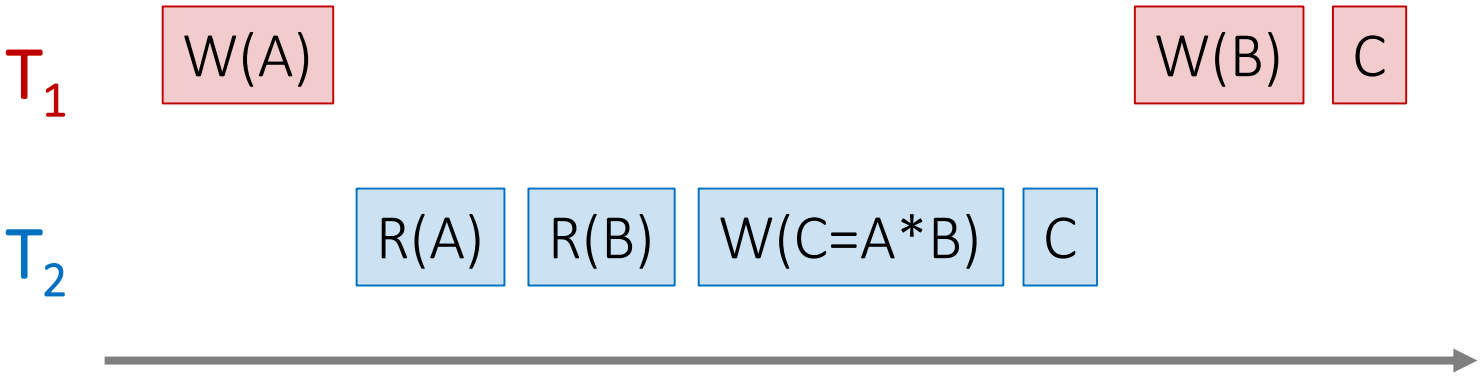
Time ↓

T1	T2
Read(X)	
X = X – 5	
Write(X)	
	Read(X) ← This reads the value of X which it should not have seen
	X = X + 5
	Write(X)
ROLLBACK	
	COMMIT

Classic Anomalies with Interleaved Execution

"Inconsistent read" / Reading partial commits:

Example:



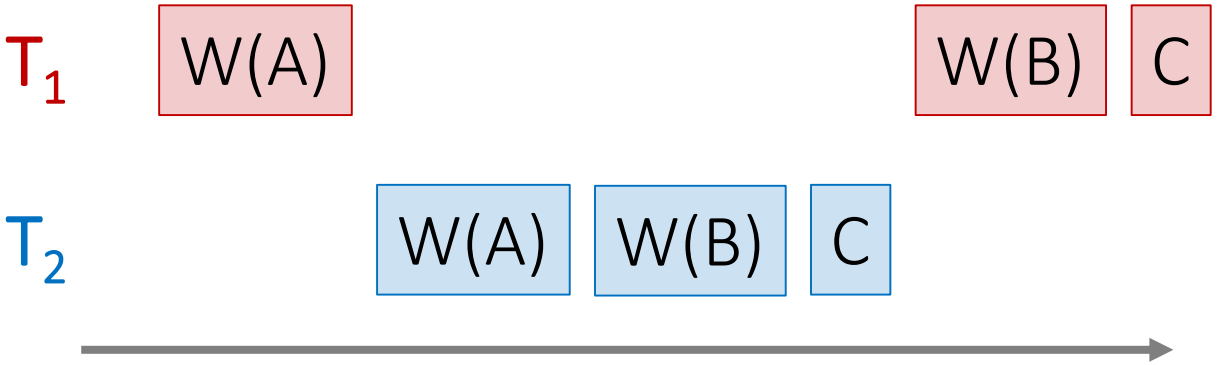
1. T_1 <u>writes</u> some data to A
2. T_2 <u>reads</u> from A *and B*, and then writes some value which depends on A & B
3. T_1 then writes to B- *now T_2's result is based on an incomplete commit*

*Again, occurring because of a **WR conflict***

Classic Anomalies with Interleaved Execution

Lost update:

Example:



1. T_1 *blind* writes some data to A
2. T_2 *blind* writes to A *and B*
3. T_1 then *blind* writes to B; now we have T_2's value for B and T_1's value for A- ***not equivalent to any serial schedule!***

*Occurring because of a **WW conflict***

Lost Update

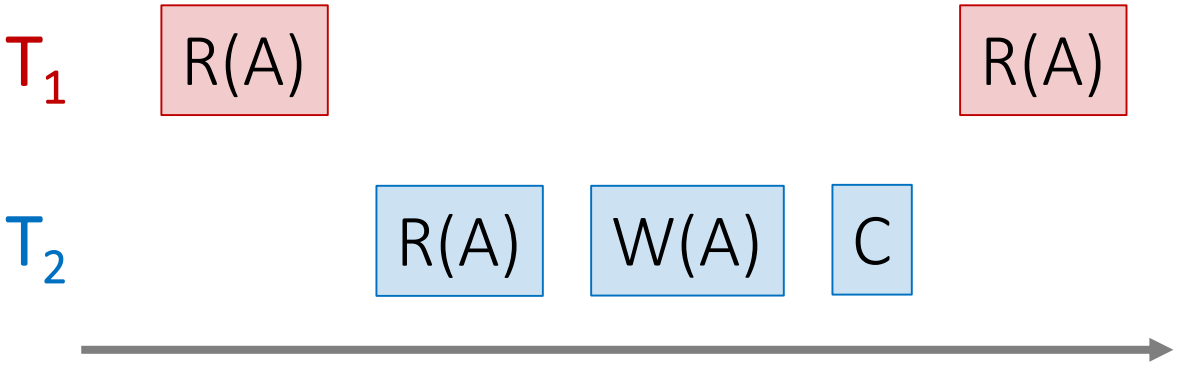
Time

T1	T2
Read(X) X = X - 5	
	Read(X) X = X + 5
This update gets lost → Write(X)	
	Write(X) ← Only this update succeeds
COMMIT	
	COMMIT

Classic Anomalies with Interleaved Execution

"Unrepeatable (or, non-repeatable) read":

Example:



1. T_1 <u>reads</u> some data from A
2. T_2 <u>writes</u> to A
3. Then, T_1 reads from A again *and now gets a different / inconsistent value*

*Occurring with / because of a **RW conflict***

Unrepeatable Read

Time

T1	T2
SELECT SUM (balance) FROM Accounts WHERE name = ‘Mary’	
	UPDATE Accounts SET balance = 1.05 * balance WHERE name = ‘Mary’
SELECT SUM (balance) FROM Accounts WHERE name = ‘Mary’	

does not introduce a phantom into predicate name=‘Mary’

It will return different value

Phantom Read

Time

T1	T2
SELECT * FROM Departments ORDER BY DepartmentID;	
	INSERT INTO Departments VALUES (600, 'Foreign Sales');
SELECT * FROM Departments ORDER BY DepartmentID;	

A new (phantom) row appears here

SQL Isolation Levels

- **READ UNCOMMITTED – dirty reads, non-repeatable reads, and phantoms allowed**
- **READ COMMITTED - dirty reads not allowed, but non-repeatable reads and phantoms allowed**
- **REPEATABLE READ – dirty reads, non-repeatable reads not allowed, but phantoms allowed**
- **SERIALIZABLE – dirty reads, non-repeatable reads, and phantoms not allowed; all schedules must be serializable**

MYSQL: Set Transaction Syntax

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic:
    ISOLATION LEVEL level
  | READ WRITE
  | READ ONLY

level:
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
```

https://dev.mysql.com/doc/refman/5.7/en/set-transaction.html

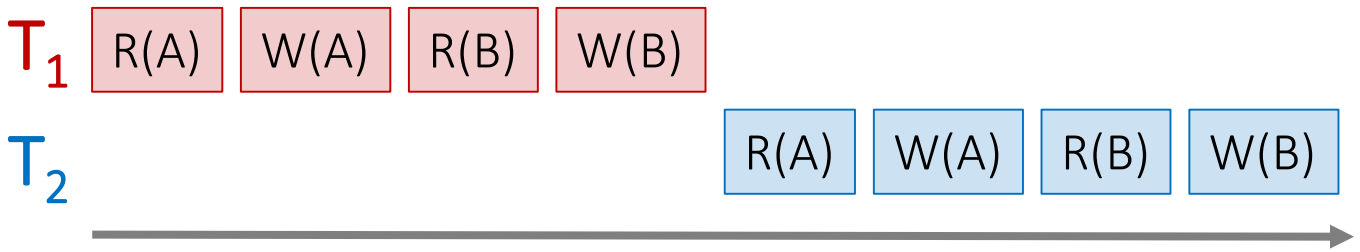
5. Conflict Serializability, Locking & Deadlock

What you will learn about in this section

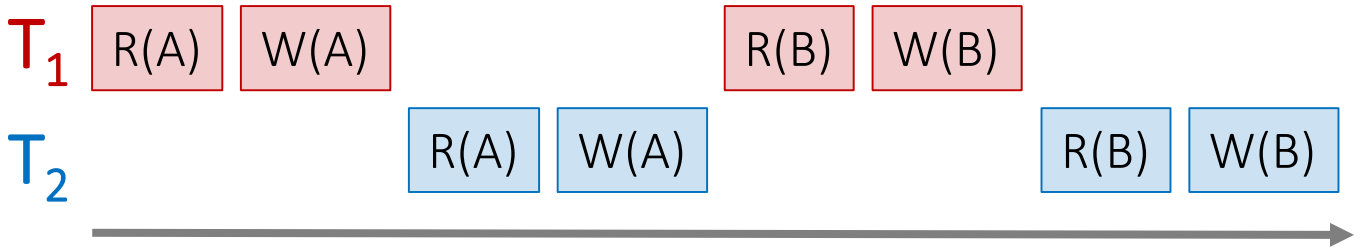
1. RECAP: Concurrency
2. Conflict Serializability
3. DAGs & Topological Orderings
4. Strict 2PL
5. Deadlocks

Recall: Concurrency as Interleaving TXNs

Serial Schedule:



Interleaved Schedule:

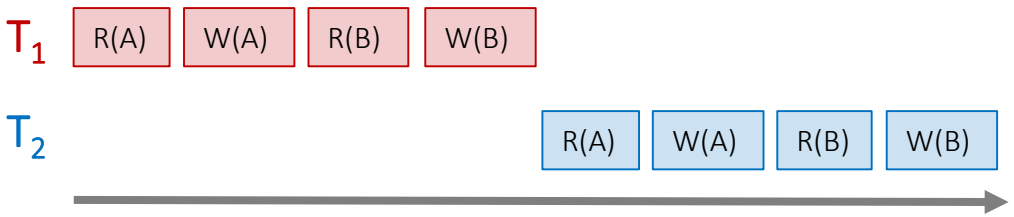


- For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

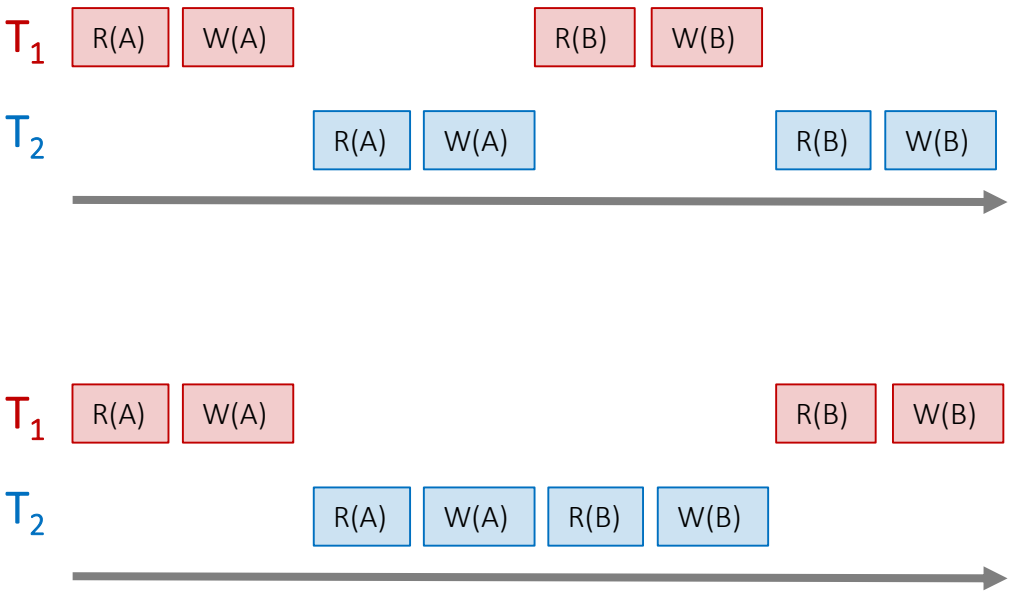
We call the particular order of interleaving a <u>schedule</u>

Recall: “Good” vs. “bad” schedules

Serial Schedule:



Interleaved Schedules:



X

Why?

We want to develop ways of discerning “good” vs. “bad” schedules

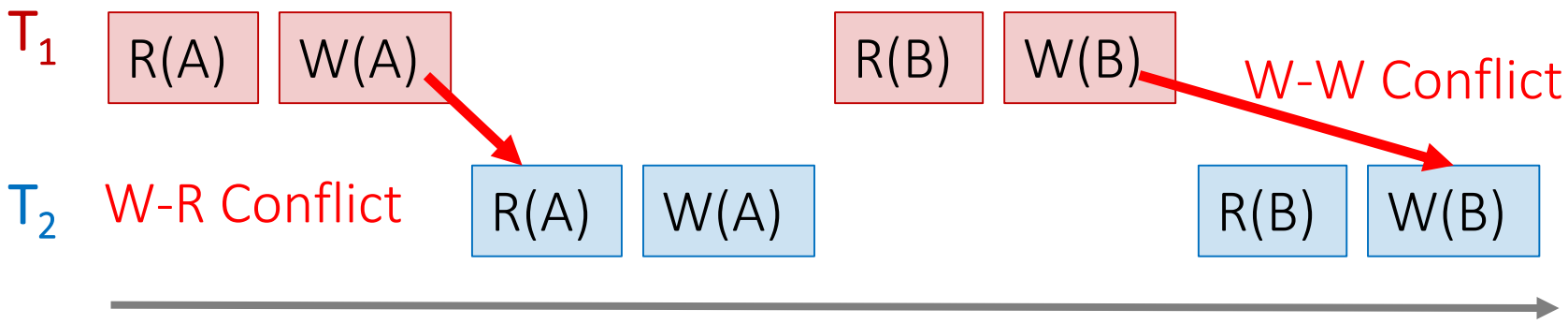
Ways of Defining "Good" vs. "Bad" Schedules

- Recall from last time: we call a schedule ***serializable*** if it is equivalent to *some* serial schedule
  - We used this as a notion of a "good" interleaved schedule, since **a serializable schedule will maintain isolation & consistency**
- Now, we'll define a stricter, but very useful variant:
  - ***<u>Conflict serializability</u>***

We'll need to define *conflicts* first..

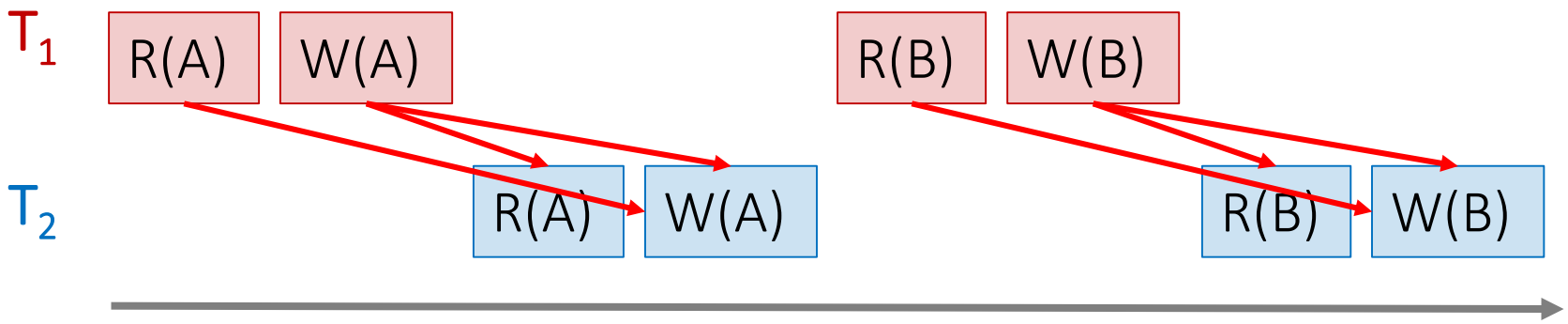
Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



All "conflicts"!

Conflict Serializability

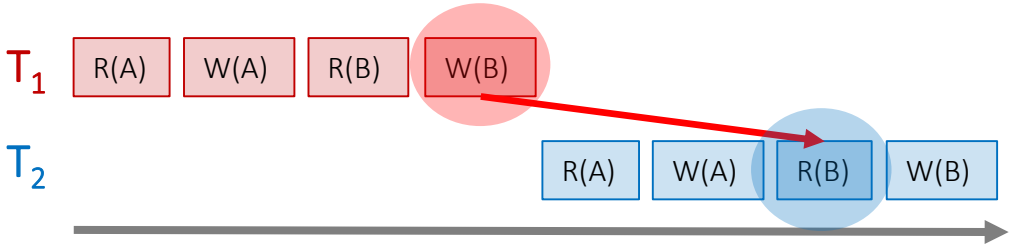
- Two schedules are **conflict equivalent** if:
  - They involve *the same actions of the same TXNs*
  - Every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

Conflict serializable ⇒ serializable

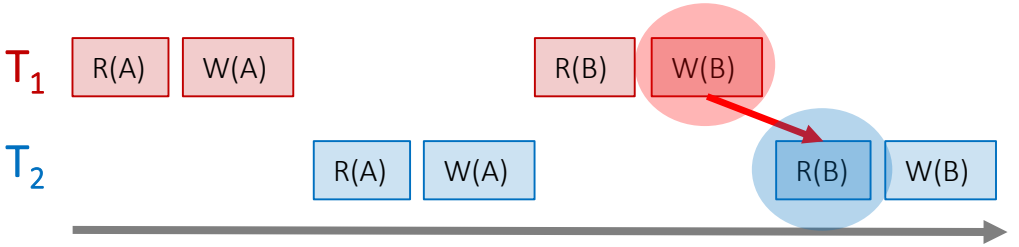
So if we have conflict serializable, we have consistency & isolation!

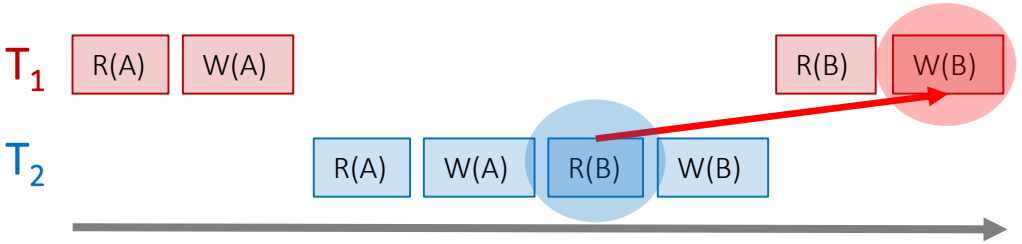
Recall: “Good” vs. “bad” schedules

Serial Schedule:



Interleaved Schedules:





Note that in the “bad” schedule, the ***order of conflicting actions is different than the above (or any) serial schedule!***

Conflict serializability also provides us with an operative notion of “good” vs. “bad” schedules!

Note: Conflicts vs. Anomalies

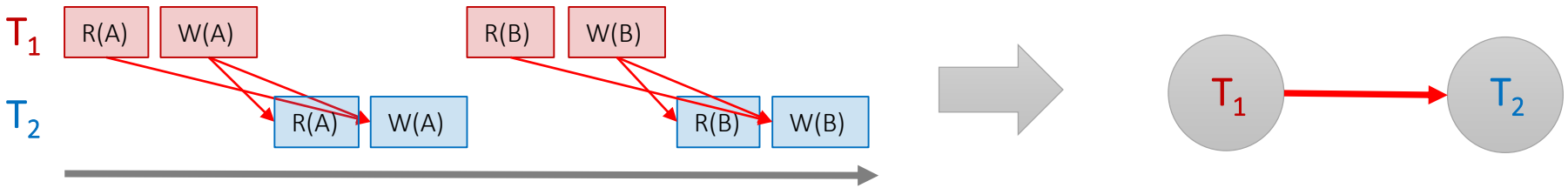
- **Conflicts** are things we talk about to help us characterize different schedules
  - Present in both “good” and “bad” schedules

- **Anomalies** are instances where isolation and/or consistency is broken because of a “bad” schedule
  - We often characterize different anomaly types by what types of conflicts predicated them

The Conflict Graph

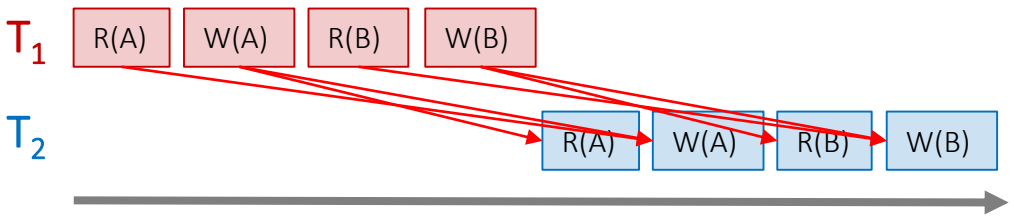
- Let's now consider looking at conflicts **at the TXN level**

- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if any actions in T_i <u>precede and conflict with</u> any actions in T_j**

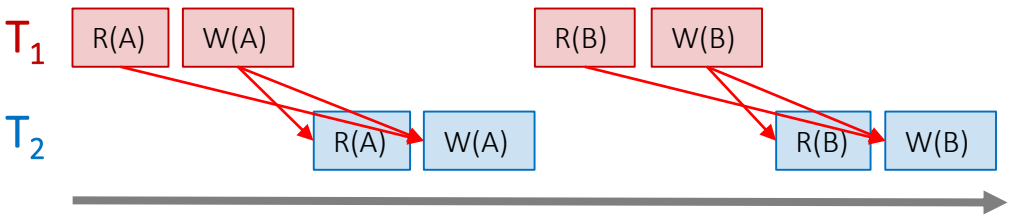


What can we say about "good" vs. "bad" conflict graphs?

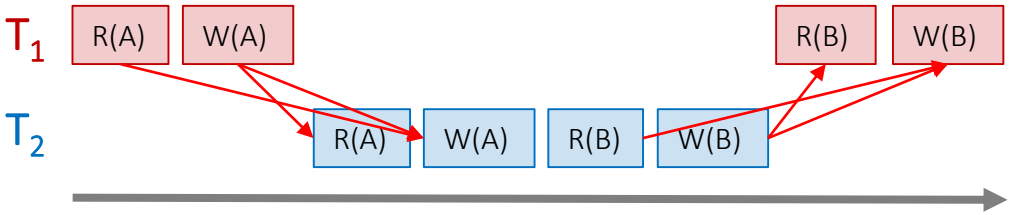
Serial Schedule:



Interleaved Schedules:



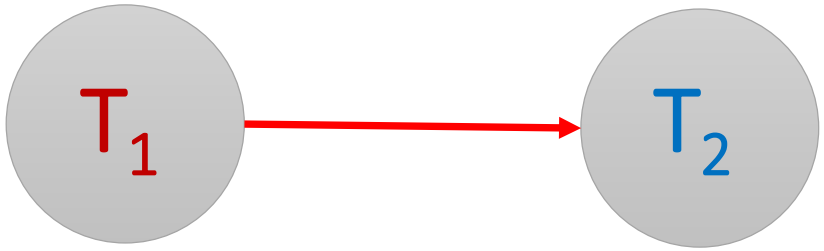
X



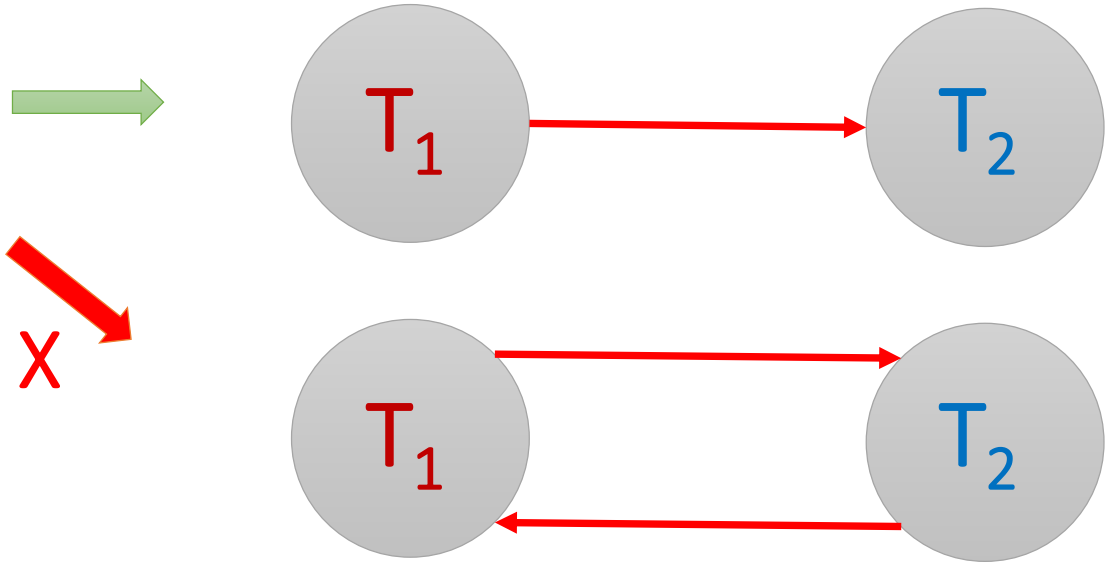
A bit complicated…

What can we say about "good" vs. "bad" conflict graphs?

Serial Schedule:



Interleaved Schedules:



Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

Let’s unpack this notion of acyclic conflict graphs…

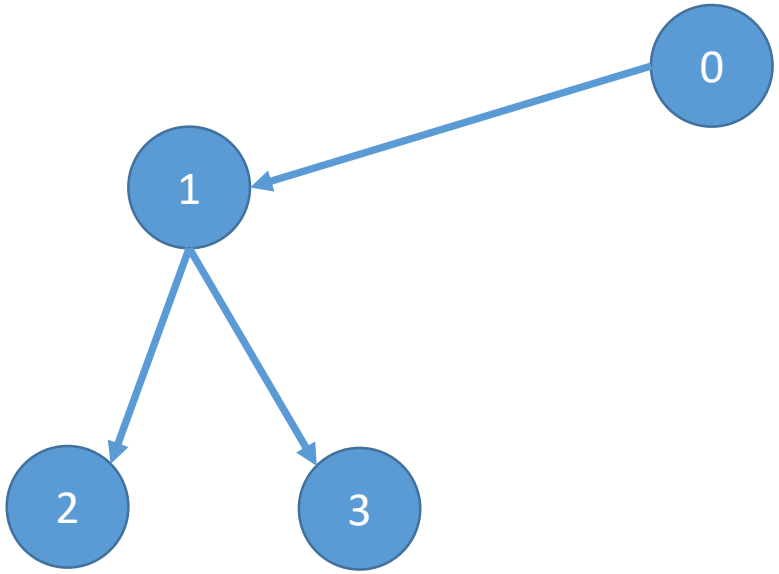
DAGs & Topological Orderings

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges

- A directed **acyclic** graph (DAG) always has one or more **topological orderings**
  - (And there exists a topological ordering *if and only if* there are no directed cycles)

DAGs & Topological Orderings

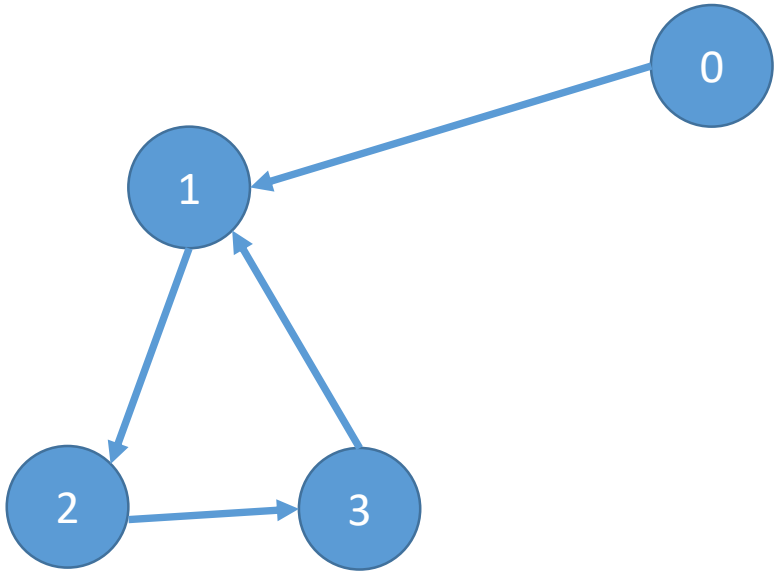
- Ex: What is one possible topological ordering here?



Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)

DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



There is none!

Connection to conflict serializability

- In the conflict graph, a topological ordering of nodes corresponds to **a serial ordering of TXNs**

- Thus an **acyclic** conflict graph → conflict serializable!

Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

Strict Two-Phase Locking

- We consider **locking**- specifically, *strict two-phase locking*- as a way to deal with concurrency, because it **guarantees conflict serializability (if it completes- see upcoming…)**

- Also (*conceptually*) straightforward to implement, and transparent to the user!

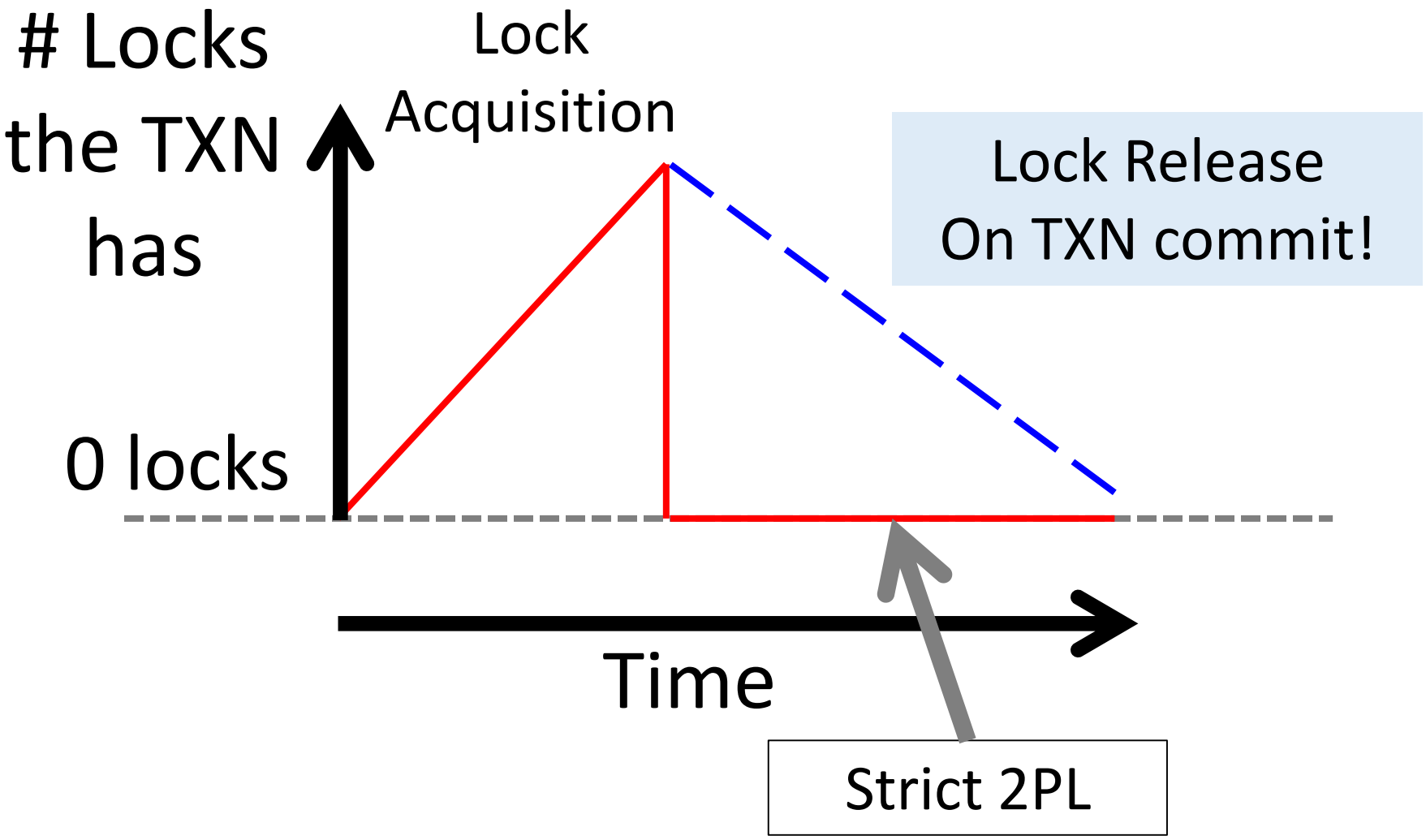
Strict Two-phase Locking (Strict 2PL) Protocol:

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object
- All locks held by a TXN are released when TXN completes.

Note: Terminology here- "exclusive", "shared"- meant to be intuitive- no tricks!

Picture of 2-Phase Locking (2PL)



Strict 2PL

<u>Theorem:</u> Strict 2PL allows only schedules whose dependency graph is acyclic

Proof Intuition: In strict 2PL, if there is an edge $T_i \rightarrow T_j$ (i.e. T_i and T_j conflict) then T_j needs to wait until T_i is finished – so *cannot* have an edge $T_j \rightarrow T_i$

Therefore, Strict 2PL only allows conflict serializable ⇒ serializable schedules

Strict 2PL

- If a schedule follows strict 2PL and locking, it is conflict serializable…
  - …and thus serializable
  - …and thus maintains isolation & consistency!
- Not all serializable schedules are allowed by strict 2PL.
- So let’s use strict 2PL, what could go wrong?

Deadlock Detection: Example

Waits-for graph:

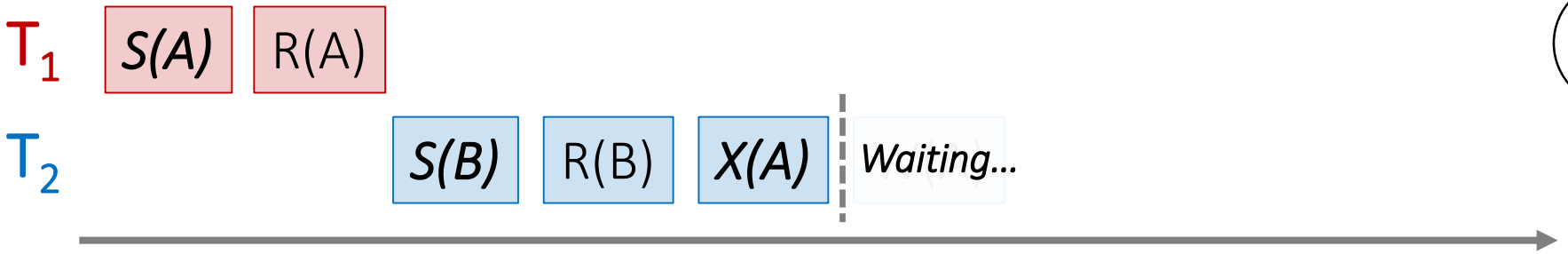
First, T_1 requests a shared lock on A to read from it

Deadlock Detection: Example

Waits-for graph:

Next, T_2 requests a shared lock on B to read from it

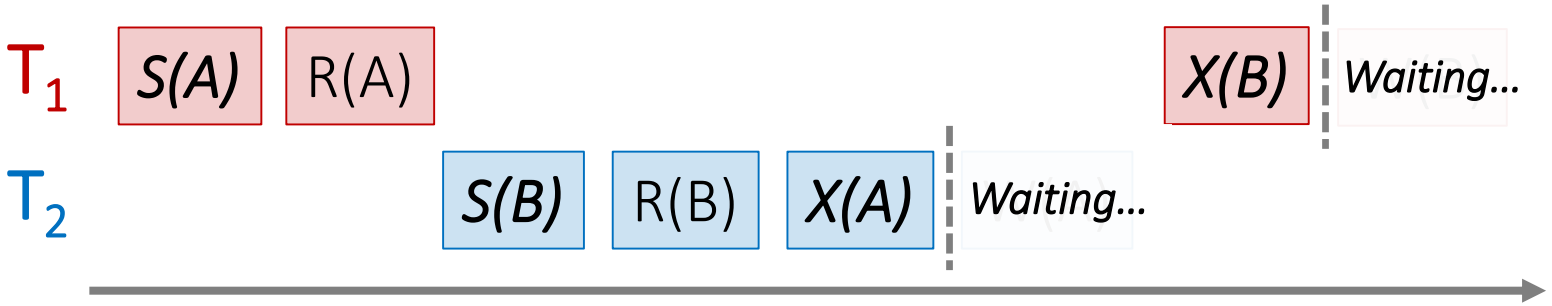
Deadlock Detection: Example



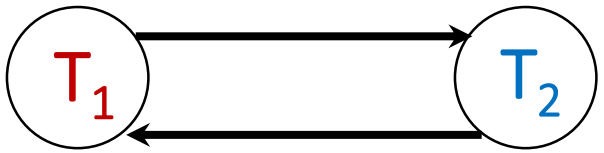
Waits-for graph:

T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1...**

Deadlock Detection: Example



Waits-for graph:



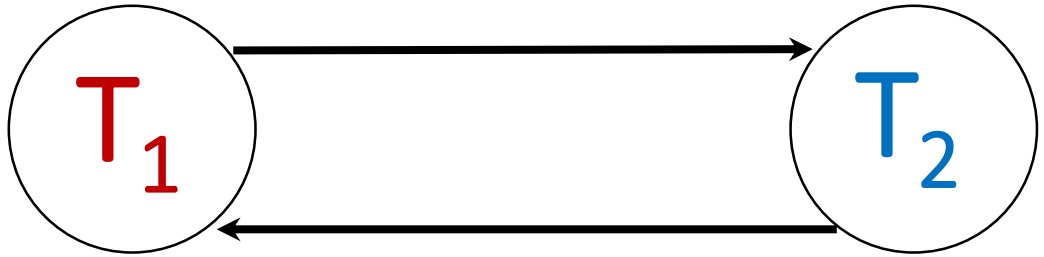
Cycle = DEADLOCK

Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2... DEADLOCK!**

```
sqlite3.OperationalError: database is locked
```

```
ERROR:  deadlock detected
DETAIL:  Process 321 waits for ExclusiveLock on tuple of
relation 20 of database 12002; blocked by process 4924.
Process 404 waits for ShareLock on transaction 689; blocked
by process 552.
HINT:  See server log for query details.
```

Deadlock!!!



Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.

- Two ways of dealing with deadlocks:

  1. Deadlock prevention

  2. Deadlock detection

Deadlock Detection

- Create the **waits-for graph**:
  - Nodes are transactions
  - There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*
- Periodically check for (***and break***) cycles in the waits-for graph

Summary

- Concurrency achieved by **interleaving TXNs** such that **isolation** & **consistency** are maintained
  - We formalized a notion of **<u>serializability</u>** that captured such a "good" interleaving schedule
- We defined **<u>conflict serializability</u>**, which implies serializability
- **Locking** allows only conflict serializable schedules
  - If the schedule completes… (it may deadlock!)

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the *CS145 Introduction to Databases* lecture given by *Christopher Ré* at *Stanford University* (http://web.stanford.edu/class/cs145/).

Transactions

- A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.

Transactions

- A transaction is a ‘logical unit of work’ on a database
  - Each transaction does something in the database
  - No part of it alone achieves anything of use or interest
- Transactions are the unit of recovery, consistency, and integrity as well
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity and Consistency

- Atomicity
  - Transactions are atomic – they don't have parts (conceptually)
  - can't be executed partially; it should not be detectable that they interleave with another transaction
- Consistency
  - Transactions take the database from one consistent state into another
  - In the middle of a transaction the database might not be consistent

Isolation and Durability

- Isolation
  - The effects of a transaction are not visible to other transactions until it has completed
  - From outside the transaction has either happened or not
  - To me this actually sounds like a consequence of atomicity…
- Durability
  - Once a transaction has completed, its changes are made permanent
  - Even if the system crashes, the effects of a transaction must remain in place

Example of transaction

- Transfer £50 from account A to account B

```
Read(A)
A = A - 50
Write(A)
Read(B)
B = B+50
Write(B)
```

transaction

Atomicity - shouldn't take money from A without giving it to B

Consistency - money isn't lost or gained

Isolation - other queries shouldn't see A or B change until completion

Durability - the money does not go back to A

The Transaction Manager

- The transaction manager enforces the ACID properties
  - It schedules the operations of transactions
  - COMMIT and ROLLBACK are used to ensure atomicity
- Locks or timestamps are used to ensure consistency and isolation for concurrent transactions (next lectures)
- A log is kept to ensure durability in the event of system failure (this lecture)

COMMIT and ROLLBACK

- COMMIT signals the successful end of a transaction
  - Any changes made by the transaction should be saved
  - These changes are now visible to other transactions
- ROLLBACK signals the unsuccessful end of a transaction
  - Any changes made by the transaction should be undone
  - It is now as if the transaction never existed

Recovery

- Transactions should be durable, but we cannot prevent all sorts of failures:
  - System crashes
  - Power failures
  - Disk crashes
  - User mistakes
  - Sabotage
  - Natural disasters
- Prevention is better than cure
  - Reliable OS
  - Security
  - UPS and surge protectors
  - RAID arrays
- Can’t protect against everything though

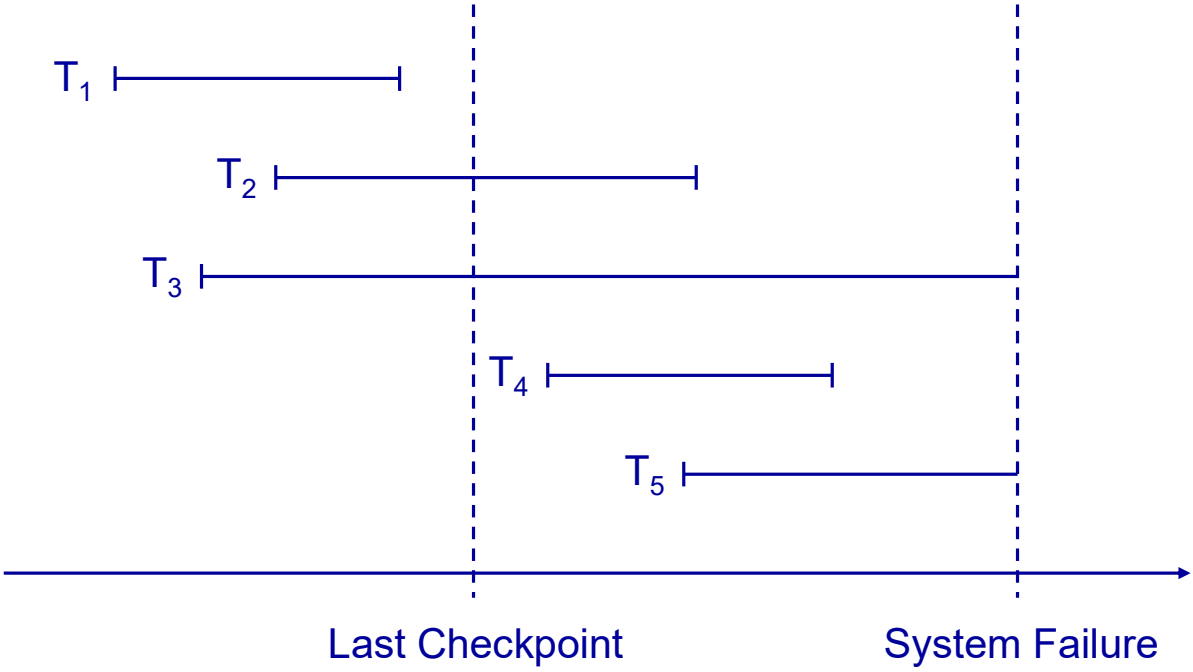
The Transaction Log

- The transaction log records the details of all transactions
  - Any changes the transaction makes to the database
  - How to undo these changes
  - When transactions complete and how
- The log is stored on disk, not in memory
  - If the system crashes it is preserved
- Write ahead log rule
  - The entry in the log must be made before COMMIT processing can complete

System Failures

- A system failure means all running transactions are affected
  - Software crashes
  - Power failures
- The physical media (disks) are not damaged
- At various times a DBMS takes a checkpoint
  - All committed transactions are written to disk
  - A record is made (on disk) of the transactions that are currently running

Types of Transactions



System Recovery

- Any transaction that was running at the time of failure needs to be undone and restarted
- Any transactions that committed since the last checkpoint need to be redone

- Transactions of type T_1 need no recovery
- Transactions of type T_3 or T_5 need to be undone and restarted
- Transactions of type T_2 or T_4 need to be redone

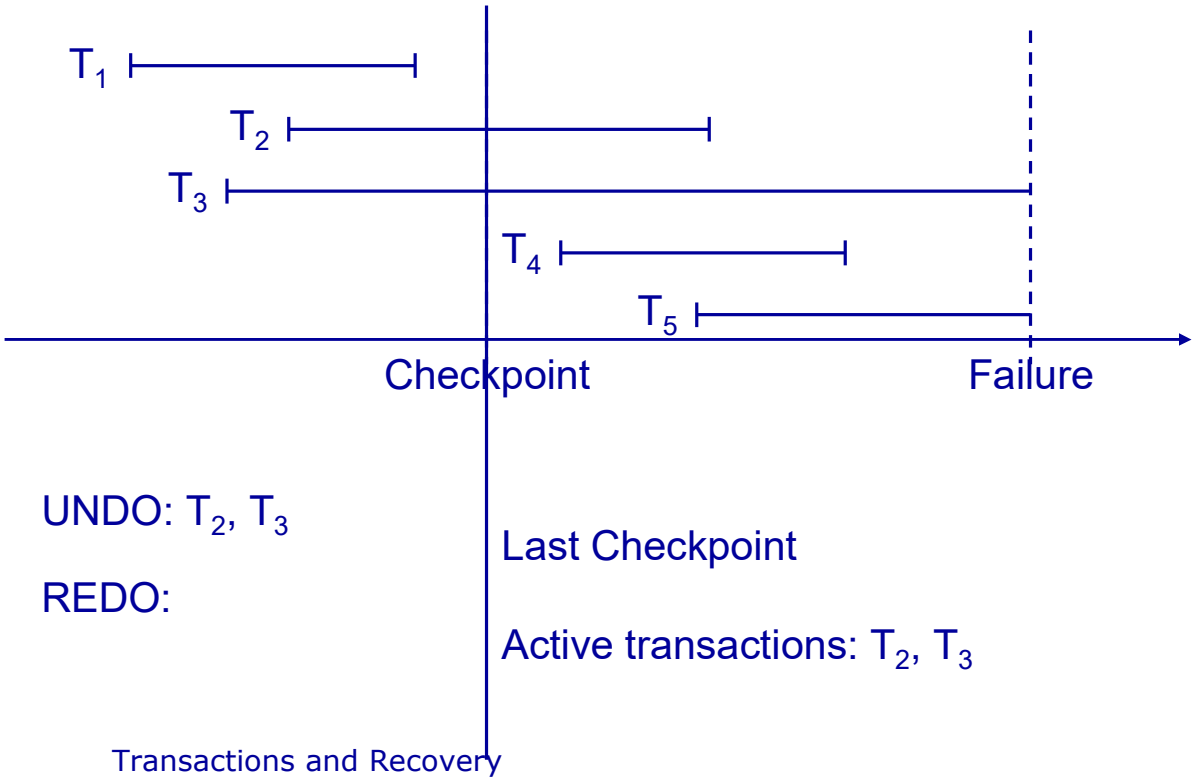
Transaction Recovery

UNDO and REDO: lists of transactions

UNDO = all transactions running at the last checkpoint
REDO = empty

For each entry in the log, starting at the last checkpoint
    If a BEGIN TRANSACTION entry is found for T
        Add T to UNDO
    If a COMMIT entry is found for T
        Move T from UNDO to REDO

Transaction Recovery



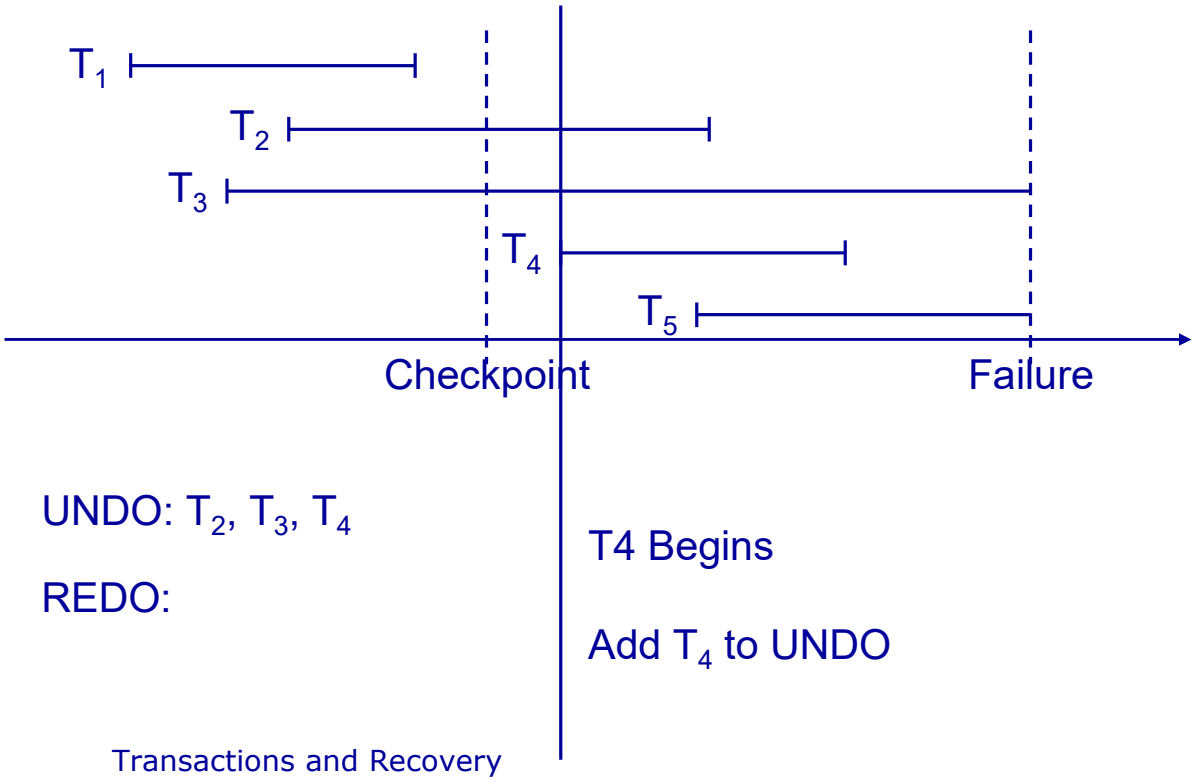
UNDO: T_2, T_3

REDO:

Last Checkpoint

Active transactions: T_2, T_3

Transaction Recovery



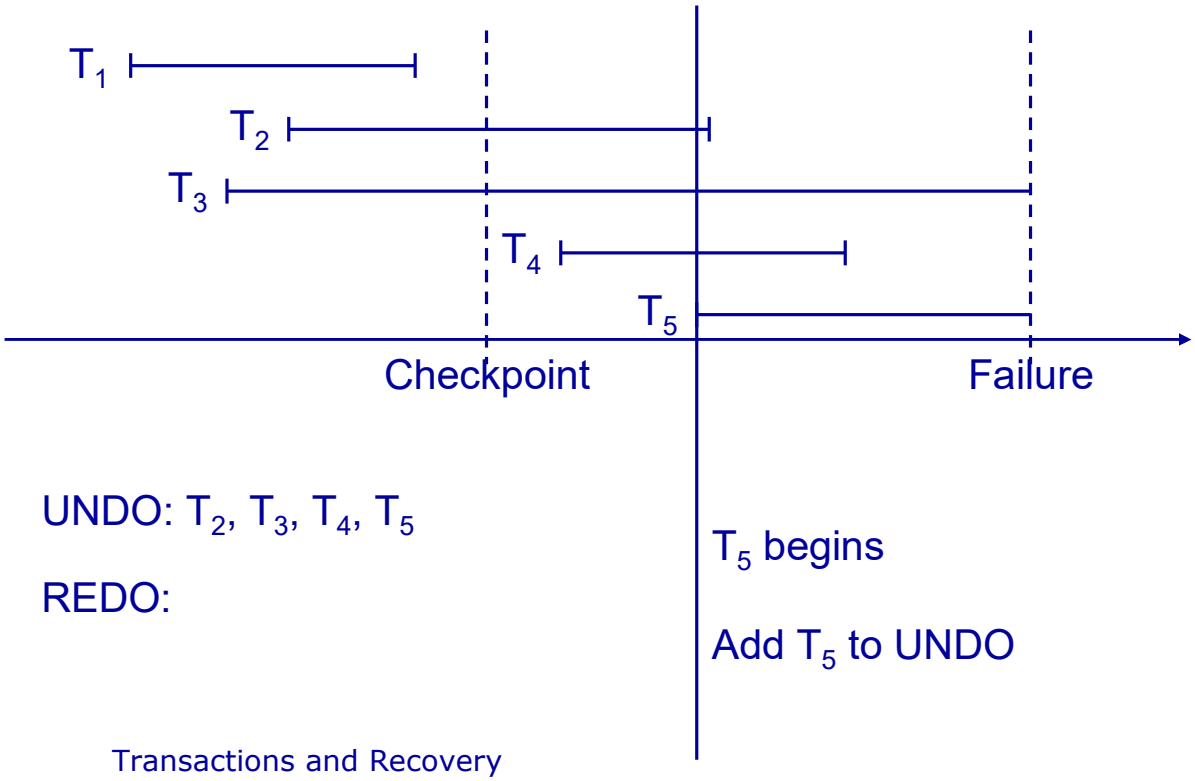
UNDO: T_2, T_3, T_4

REDO:

T4 Begins

Add T_4 to UNDO

Transaction Recovery



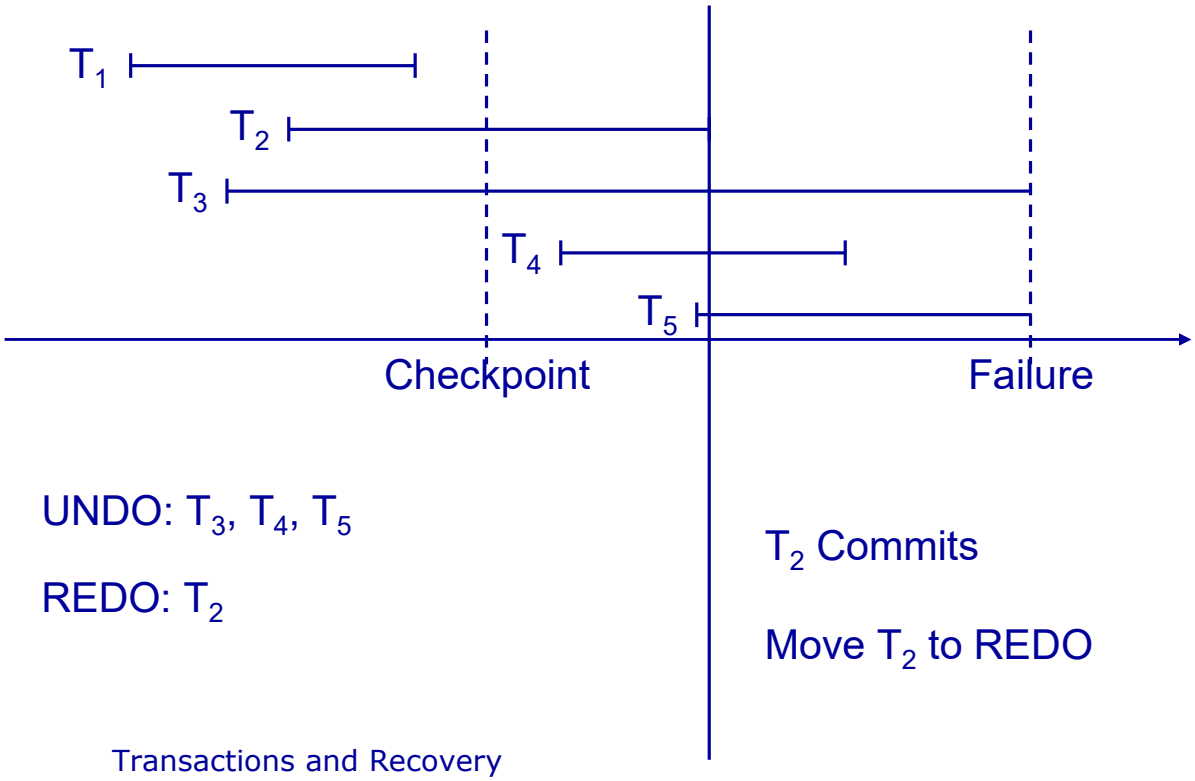
UNDO: T_2, T_3, T_4, T_5

REDO:

T_5 begins

Add T_5 to UNDO

Transaction Recovery



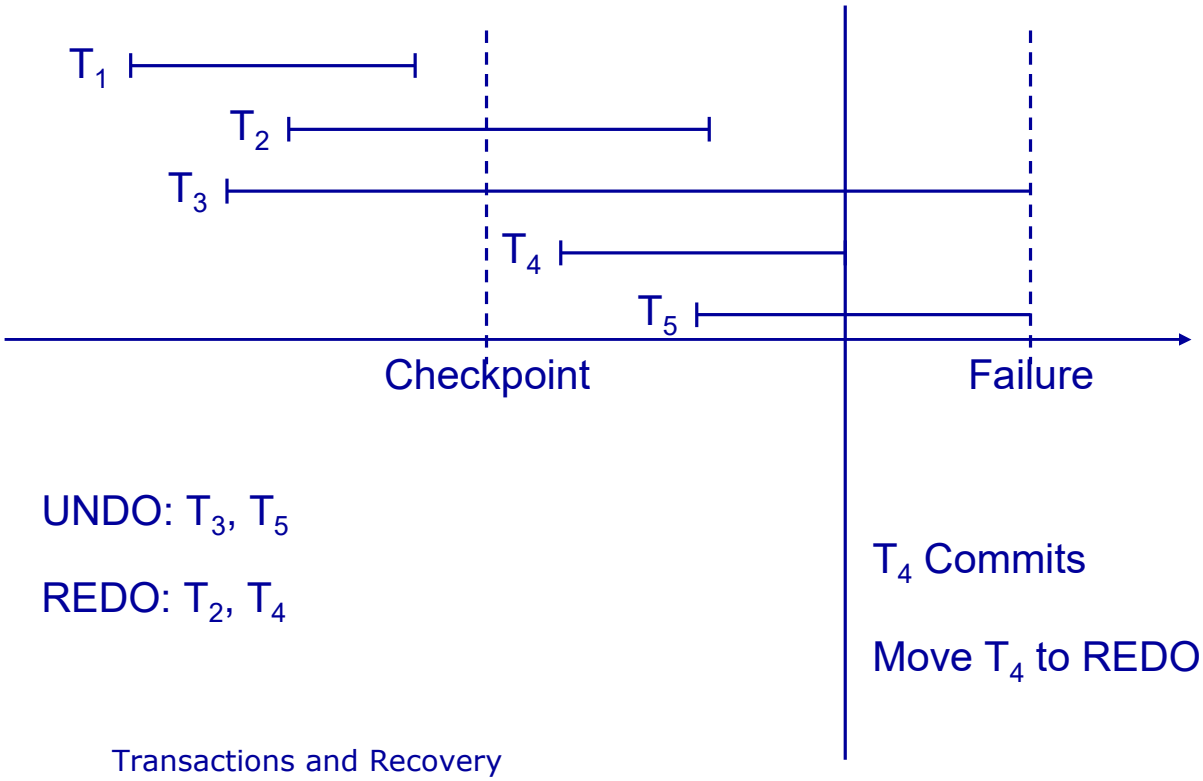
UNDO: T_3, T_4, T_5

REDO: T_2

T_2 Commits

Move T_2 to REDO

Transaction Recovery



UNDO: T_3, T_5

REDO: T_2, T_4

T_4 Commits

Move T_4 to REDO

Forwards and Backwards

- Backwards recovery
  - We need to undo some transactions
  - Working backwards through the log we undo any operation by a transaction on the UNDO list
  - This returns the database to a consistent state
- Forwards recovery
  - Some transactions need to be redone
  - Working forwards through the log we redo any operation by a transaction on the REDO list
  - This brings the database up to date

Media Failures

- System failures are not too severe
  - Only information since the last checkpoint is affected
  - This can be recovered from the transaction log
- Media failures (disk crashes etc) are more serious
  - The data stored to disk is damaged
  - The transaction log itself may be damaged

Backups

- Backups are needed to recover from media failure
  - The transaction log and entire contents of the database is written to secondary storage (often tape)
  - Time consuming, and often requires down time
- Backups frequency
  - Frequent enough that little information is lost
  - Not so frequent as to cause problems
  - Every day (night) is common
- Backup storage

Recovery from Media Failure

- Restore the database from the last backup
- Use the transaction log to redo any changes made since the last backup
- If the transaction log is damaged you can't do step 2
  - Store the log on a separate physical device to the database
  - The risk of losing both is then reduced

Concurrency

- Large databases are used by many people
  - Many transactions to be run on the database
  - It is desirable to let them run at the same time as each other
  - Need to preserve isolation
- If we don't allow for concurrency then transactions are run sequentially
  - Have a queue of transactions
  - Long transactions (eg backups) will make others wait for long periods

Concurrency Problems

- In order to run transactions concurrently we interleave their operations
- Each transaction gets a share of the computing time
- This leads to several sorts of problems
  - Lost updates
  - Uncommitted updates
  - Incorrect analysis
- All arise because isolation is broken

Lost Update

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

- T1 and T2 read X, both modify it, then both write it out
  - The net effect of T1 and T2 should be no change on X
  - Only T2's change is seen, however, so the final value of X has increased by 5

Uncommitted Update

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	X = X + 5
	Write(X)
ROLLBACK	
	COMMIT

- T2 sees the change to X made by T1, but T1 is rolled back
  - The change made by T1 is undone on rollback
  - It should be as if that change never happened

Inconsistent analysis

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	Read(Y)
	Sum = X+Y
Read(Y)	
Y = Y + 5	
Write(Y)	

- T1 doesn't change the sum of X and Y, but T2 sees a change
  - T1 consists of two parts – take 5 from X and then add 5 to Y
  - T2 sees the effect of the first, but not the second

Optimization

BBM471 Database Management Systems

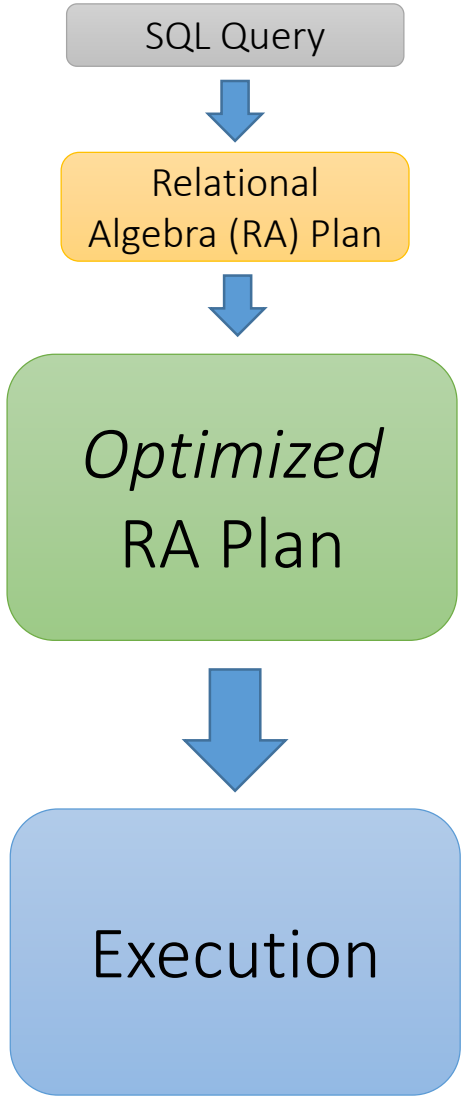
Today's Lecture

1. Logical Optimization
2. Physical Optimization

Logical vs. Physical Optimization

- **<u>Logical optimization:</u>**
  - Find equivalent plans that are more efficient
  - *Intuition: Minimize # of tuples at each step by changing the order of RA operators*

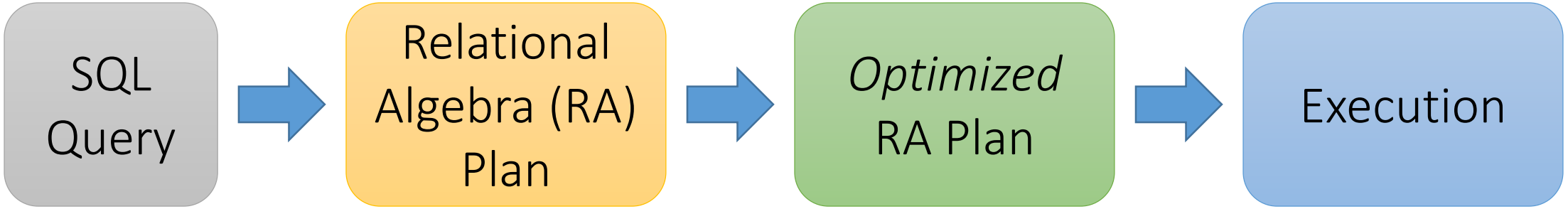
- **<u>Physical optimization:</u>**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)*



1. Logical Optimization

RDBMS Architecture

How does a SQL engine work ?



Declarative query (from user)

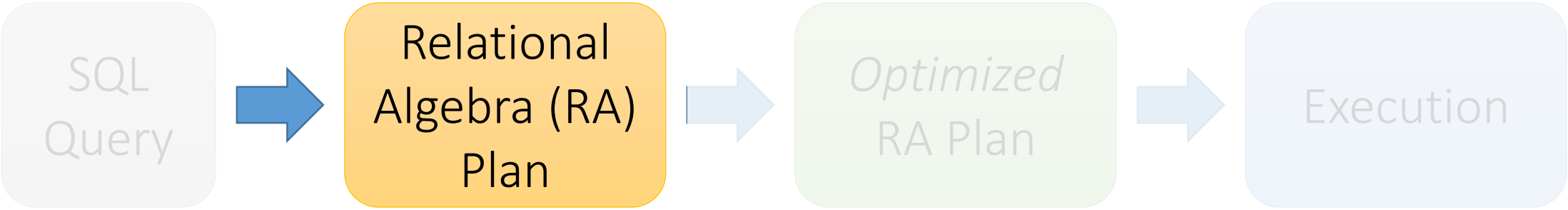
Translate to relational algebra expresson

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

RDBMS Architecture

How does a SQL engine work ?



Relational Algebra allows us to translate declarative (SQL) queries into precise and optimizable expressions!

What does DBMS Do When You submit a Query?

- Translates SQL into get/put req. to backend storage
- Extracts, processes, transforms tuples from blocks
- Performs tons of optimizations
  - Choosing algorithms for SQL operators (hashing, sorting)
  - Ordering of operators (small intermediate results)
  - Semantic rewritings of queries
  - Parallel execution and concurrency
  - Load and admission control
  - Layout of data on backend storage
  - Buffer management and caching
  - …

Input: SQL statement
Output: {tuples}

DBMS vs. OS Optimizations

- Many DBMS tasks are also carried out by OS
  - Load control
  - Buffer management
  - Access to external storage
  - Scheduling of processes

- What is the difference?
  - DBMS has intimate knowledge of workload
  - DBMS can predict and shape access pattern of a query
  - DBMS knows the contention between queries
  - OS does generic optimizations

Recall: Relational Algebra (RA)

- <u>Five **basic** operators:</u>
  1. Selection: σ
  2. Projection: Π
  3. Cartesian Product: $\times$
  4. Union: $\cup$
  5. Difference: -
- <u>Derived or auxiliary operators:</u>
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ
  - Division

Recall: Converting SFW Query -> RA

Students(sid,name,gpa)
People(ssn,name,address)

```
SELECT DISTINCT
  gpa,
  address
FROM Students S,
     People P
WHERE gpa > 3.5 AND
   s.name = p.name;
```

$$\Pi_{gpa,address}(\sigma_{gpa>3.5}(S \bowtie P))$$

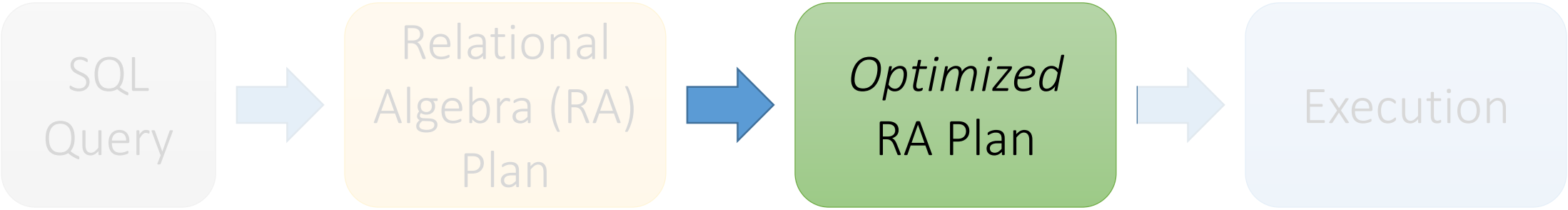
How do we represent this query in RA?

Recall: Logical Equivalence of RA Plans

- Given relations R(A,B) and S(B,C):
  - Here, projection & selection commute:
    - $\sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R))$
  - What about here?
    - $\sigma_{A=5}(\Pi_B(R)) \,?= \Pi_B(\sigma_{A=5}(R))$

RDBMS Architecture

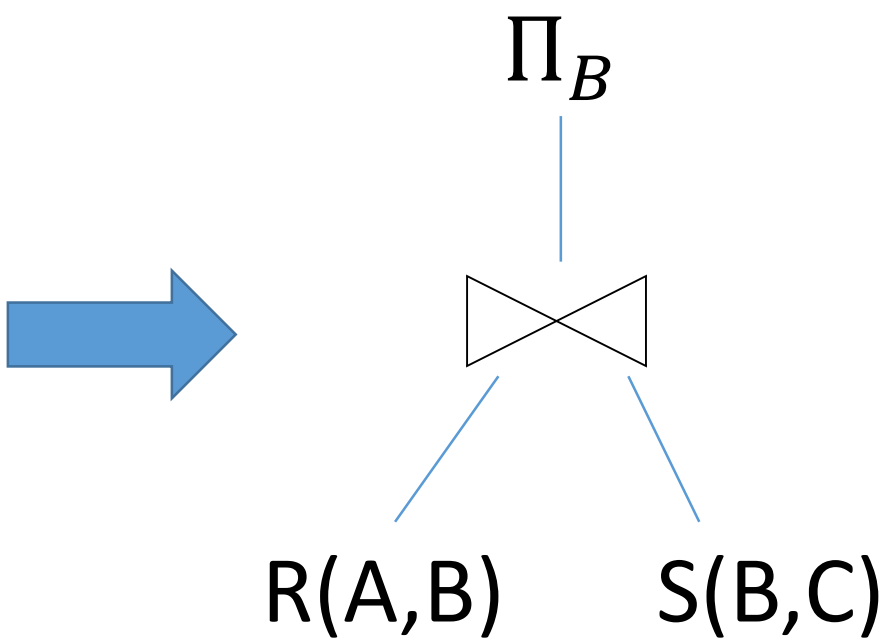
How does a SQL engine work ?



We'll look at how to then optimize these plans now

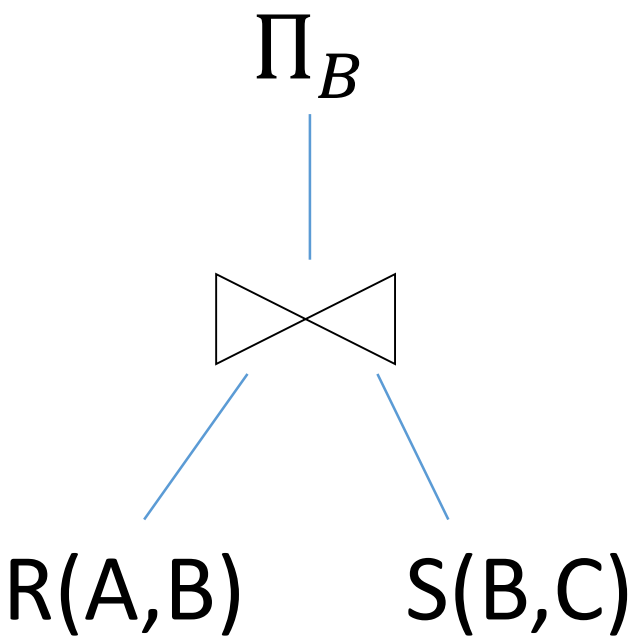
Note: We can visualize the plan as a tree

$$\Pi_B(R(A,B) \bowtie S(B,C))$$



Bottom-up tree traversal = order of operation execution!

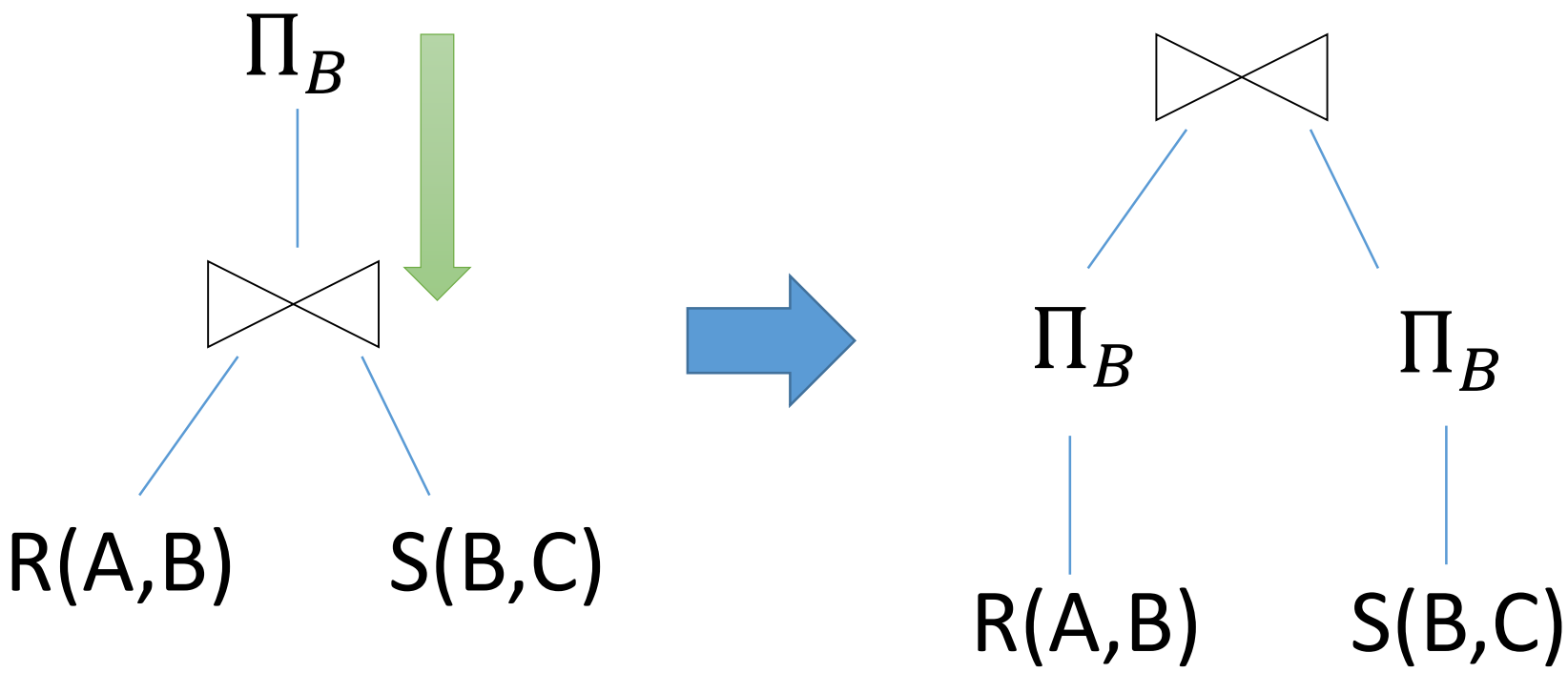
A simple plan



What SQL query does this correspond to?

Are there any logically equivalent RA expressions?

“Pushing down” projection



Why might we prefer this plan?

Takeaways

- This process is called logical optimization
- Many equivalent plans used to search for “good plans”
- Relational algebra is an important abstraction

RA commutators

- The basic commutators:
  - Push **projection** through **(1) selection**, **(2) join**
  - Push **selection** through **(3) selection, (4) projection, (5) join**
  - *Also:* Joins can be re-ordered!

- Note that this is not an exhaustive set of operations
  - This covers *local re-writes; global re-writes possible but much harder*

This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!

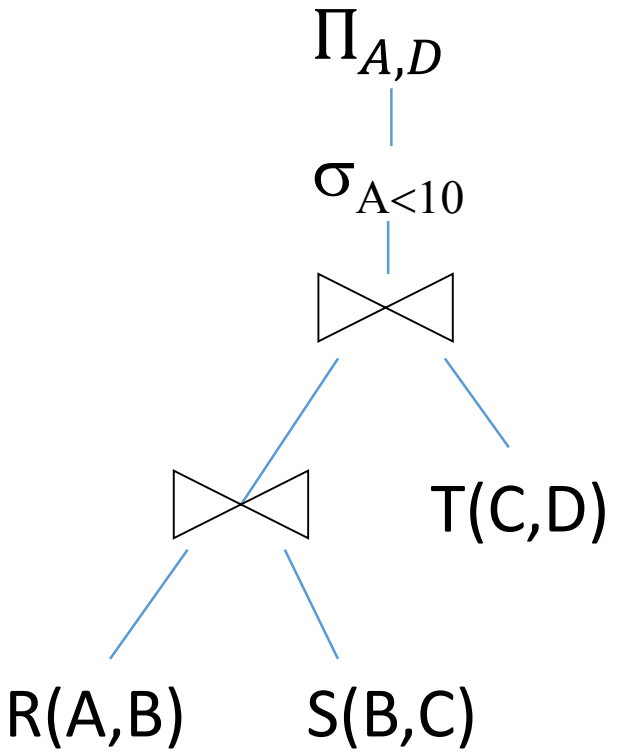
Optimizing the SFW RA Plan

Translating to RA

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$



Logical Optimization

- Heuristically, we want selections and projections to occur as early as possible in the plan
  - Terminology: “push down **selections**” and “pushing down **projections.”**

- **Intuition:** We will have fewer tuples in a plan.
  - Could fail if the selection condition is very expensive (say runs some image processing algorithm).
  - Projection could be a waste of effort, but more rarely.

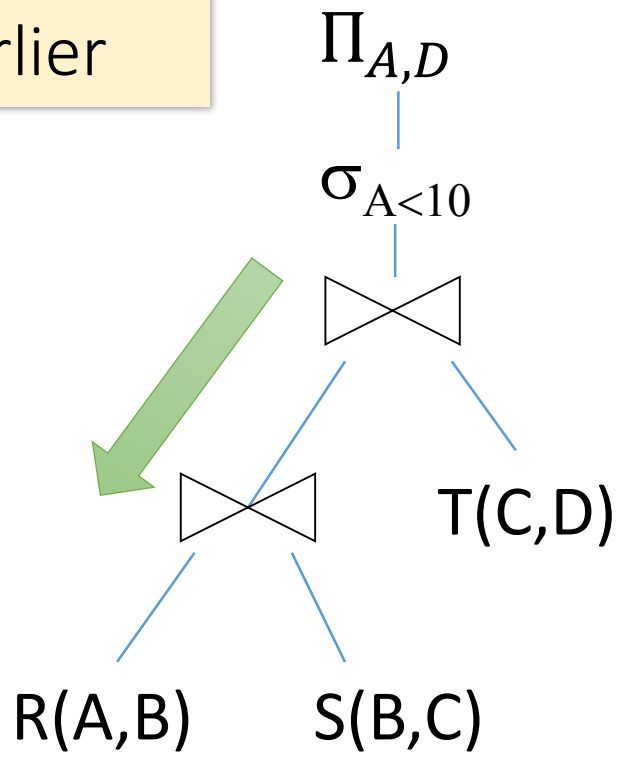
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down selection on A so it occurs earlier

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$



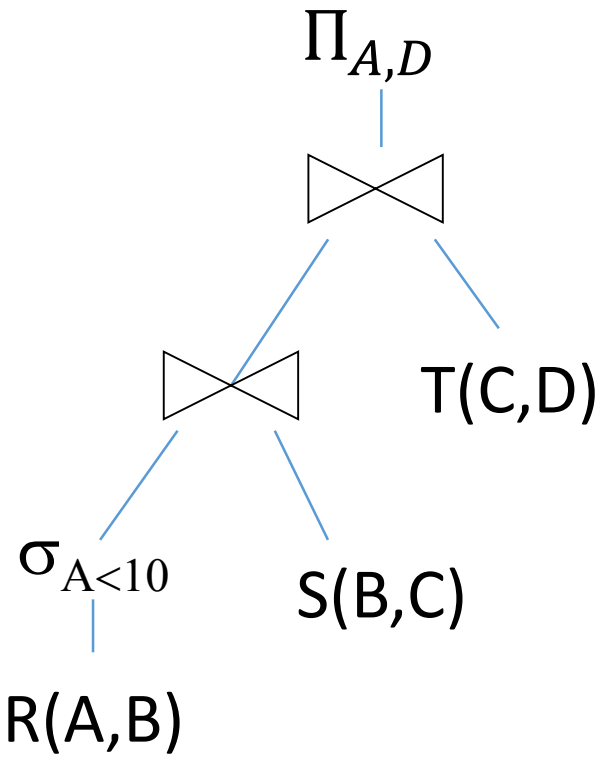
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down selection on A so it occurs earlier

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$



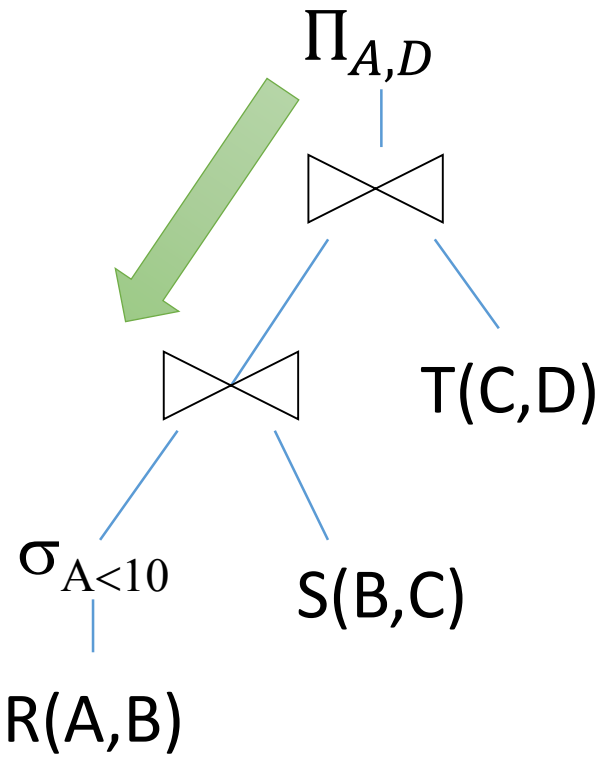
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down projection so it occurs earlier

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$



Optimizing RA Plan

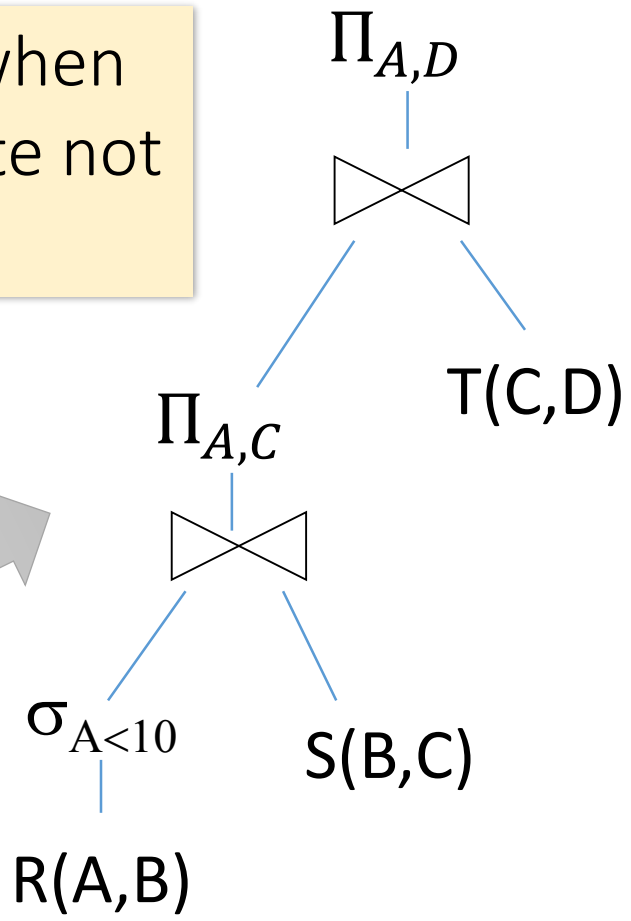
R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

We eliminate B earlier!

In general, when is an attribute not needed…?

$$\Pi_{A,D}\left(T \bowtie \Pi_{A,c}(\sigma_{A<10}(R) \bowtie S)\right)$$



Query Rewrite: Unnesting of Views

- Example: Unnesting of Views

```
select A.x
from   A
where y in
   (select y from B)
```

```
select A.x
from   A, B
where A.y = B.y
```

- Example: Unnesting of Views

```
select A.x
from   A
where exists
   (select * from B where A.y = B-y)
```

```
select A.x
from   A, B
where A.y = B.y
```

Query Rewrite: Predicate Augmentation

- Example: Predicate Augmentation

```
select *
from   A, B, C
where  A.x = B.x
 and B.x = C.x
```

```
select *
from   A, B, C
where  A.x = B.x
 and B.x = C.x
 and A.x = C.x
```

Why is that useful?

Why Predicate Augmentation?

A (odd numbers)

...	x
...	1
...	3
...	5
...	...

B (all numbers)

...	x
...	1
...	2
...	3
...	...

C (even numbers)

...	x
...	2
...	4
...	6
...	...

- Cost((A x C) x B) < Cost((A x B) x C)
  - get second join for free
- Query Rewrite does not know that, ...
  - but it knows that it might happen and hopes for optimizer...
- Codegen gets rid of unnecessary predicates (e.g., A.x = B.x)

Query Optimization

- Two tasks
  - Determine order of operators
  - Determine algorithm for each operator (hashing, sorting, ...)
- Components of a query optimizer
  - Search space
  - Cost model
  - Enumeration algorithm (NP hard)
- Working principle
  - Enumerate alternative plans
  - Apply cost model to alternative plans
  - Select plan with lowest expected cost

Optimization: Does It Really Matter? - 1

- $A \bowtie B \bowtie C$
  - size(A) = 10,000
  - size(B) = 100
  - size(C) = 1
  - cost($X \bowtie Y$) = size(X) + size(Y)

- cost($(A \bowtie B) \bowtie C$) = 1,010,101
  - cost($A \bowtie B$) = 10,100
  - cost($\mathbf{X} \bowtie C$) = 1,000,001 with **X = A ⋈ B**

- cost ($A \bowtie (B \bowtie C)$) = 10,201
  - cost($B \bowtie C$) = 101
  - cost($A \bowtie \mathbf{X}$) = 10,100 with **X = B ⋈ C**

Optimization: Does It Really Matter? - 2

- $A \bowtie B \bowtie C$
  - size(A) = 1000
  - size(B) = 1
  - size(C) = 1
  - cost($X \bowtie Y$) = size(X) * size(Y)

- cost($(A \bowtie B) \bowtie C$) = 2000
  - cost($A \bowtie B$) = 1000
  - cost($\mathbf{X} \bowtie C$) = 1000 with $\mathbf{X = A \bowtie B}$

- cost ($A \bowtie (B \bowtie C)$) = 1001
  - cost($B \bowtie C$) = 1
  - cost($A \bowtie \mathbf{X}$) = 1000 with $\mathbf{X = B \bowtie C}$

2. Physical Optimization

What you will learn about in this section

1. Index Selection
2. Histograms
3. Materialized Views

Index Selection

Input:

- Schema of the database
- **Workload description:** set of (query template, frequency) pairs

Goal: Select a set of indexes that minimize execution time of the workload.

- Cost / benefit balance: Each additional index may help with some queries, but requires updating

This is an optimization problem!

Example

Workload description:

```
SELECT pname
FROM Product
WHERE year = ? AND category = ?
```

Frequency
10,000,000

```
SELECT pname,
FROM Product
WHERE year = ? AND Category = ?
AND manufacturer = ?
```

Frequency
10,000,000

Which indexes might we choose?

Example

Workload description:

```
SELECT pname
FROM Product
WHERE year = ? AND category =?
```

Frequency 10,000,000

```
SELECT pname
FROM Product
WHERE year = ? AND Category =?
AND manufacturer = ?
```

Frequency 100

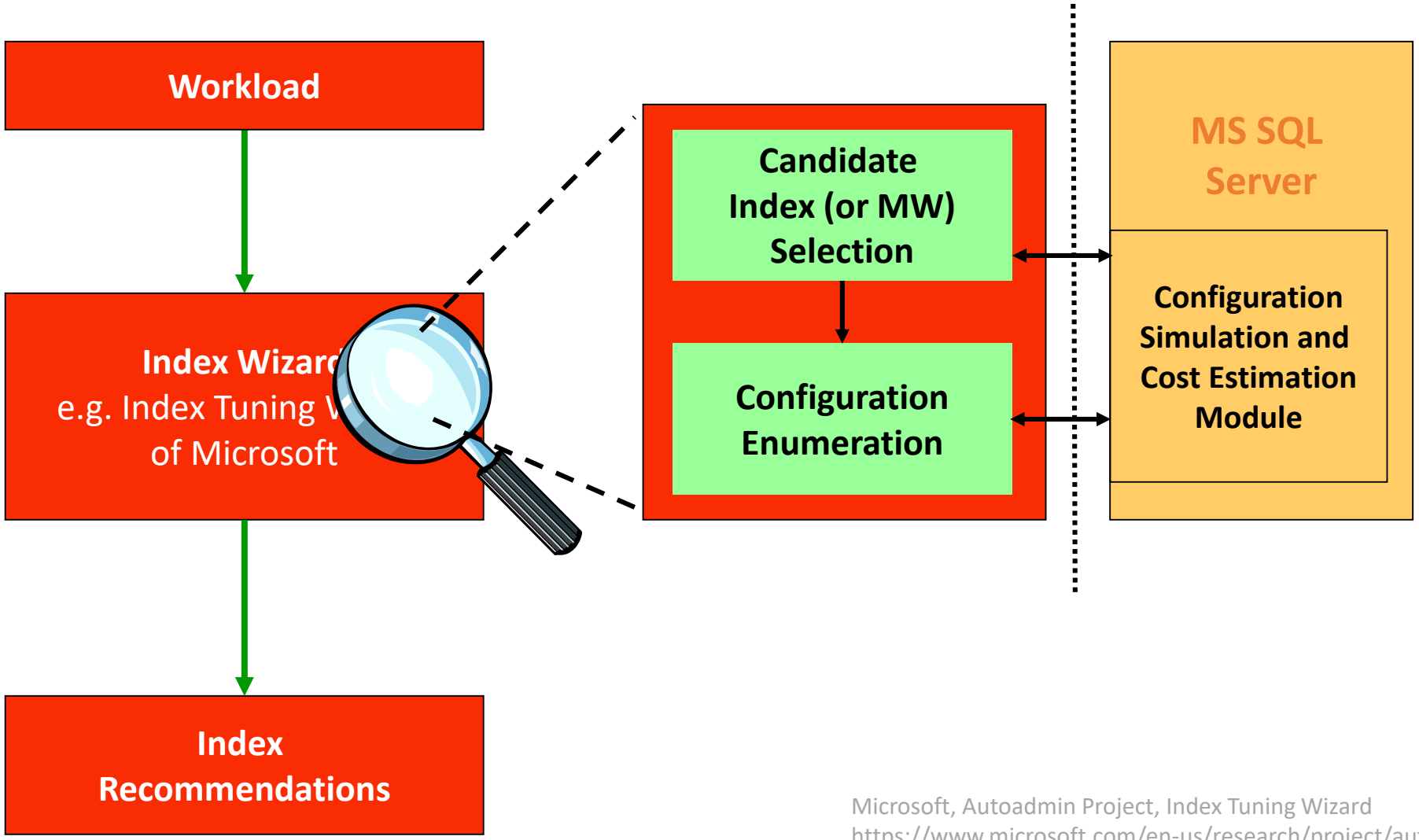
Now which indexes might we choose? Worth keeping an index with manufacturer in its search key around?

Simple Heuristic

- Can be framed as standard optimization problem: Estimate how cost changes when we add index.
    - We can ask the optimizer!
- Search over all possible space is too expensive, optimization surface is really nasty.
    - Real DBs may have 1000s of tables!
- Techniques to exploit *structure* of the space.
    - In SQL Server Autoadmin.

NP-hard problem, but can be solved!

Automatic Database Design (Physical Layout)



Microsoft, Autoadmin Project, Index Tuning Wizard
https://www.microsoft.com/en-us/research/project/autoadmin/

Estimating index cost?

- Note that to frame as optimization problem, we first need an estimate of the ***cost*** of an index lookup

- Need to be able to estimate the costs of different indexes / index types…

We will see this mainly depends on getting estimates of result set size!

Histograms & IO Cost Estimation

IO Cost Estimation via Histograms

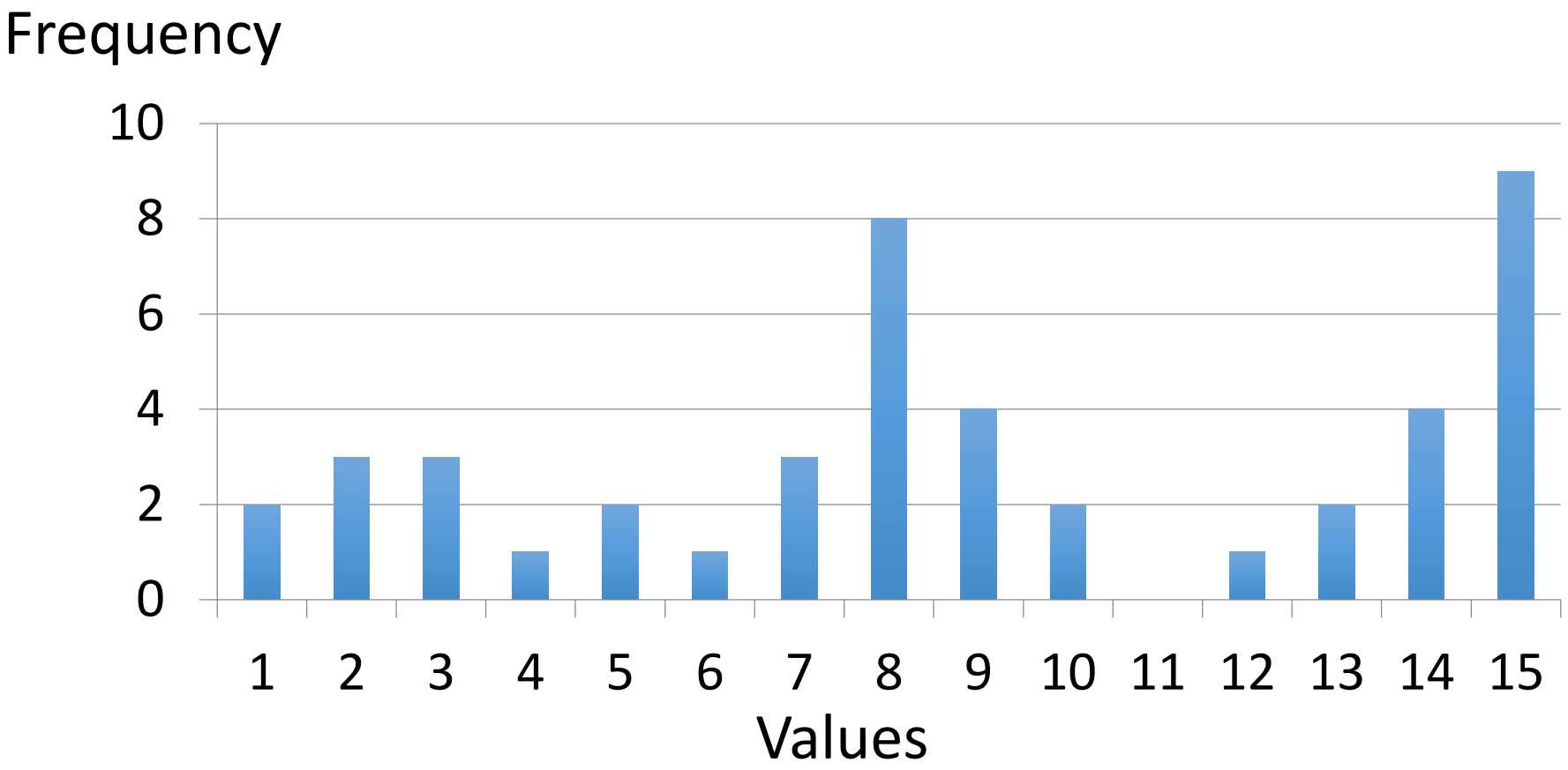
- For **index selection**:
  - What is the cost of an index lookup?
- Also for **deciding which algorithm to use**:
  - Ex: To execute $\mathrm{R} \bowtie S$, which join algorithm should DBMS use?
  - **What if we want to compute $\boldsymbol{\sigma_{A>10}(\mathrm{R}) \bowtie \sigma_{B=1}(S)}$?**
- In general, we will need some way to ***estimate intermediate result set sizes***

Histograms provide a way to efficiently store estimates of these quantities

Histograms

- A histogram is a set of value ranges (“buckets”) and the frequencies of values in those buckets occurring
- Can be used to estimate cardinality of result sets
- How to choose the buckets?
  - Equiwidth & Equidepth
- Turns out high-frequency values are **very** important

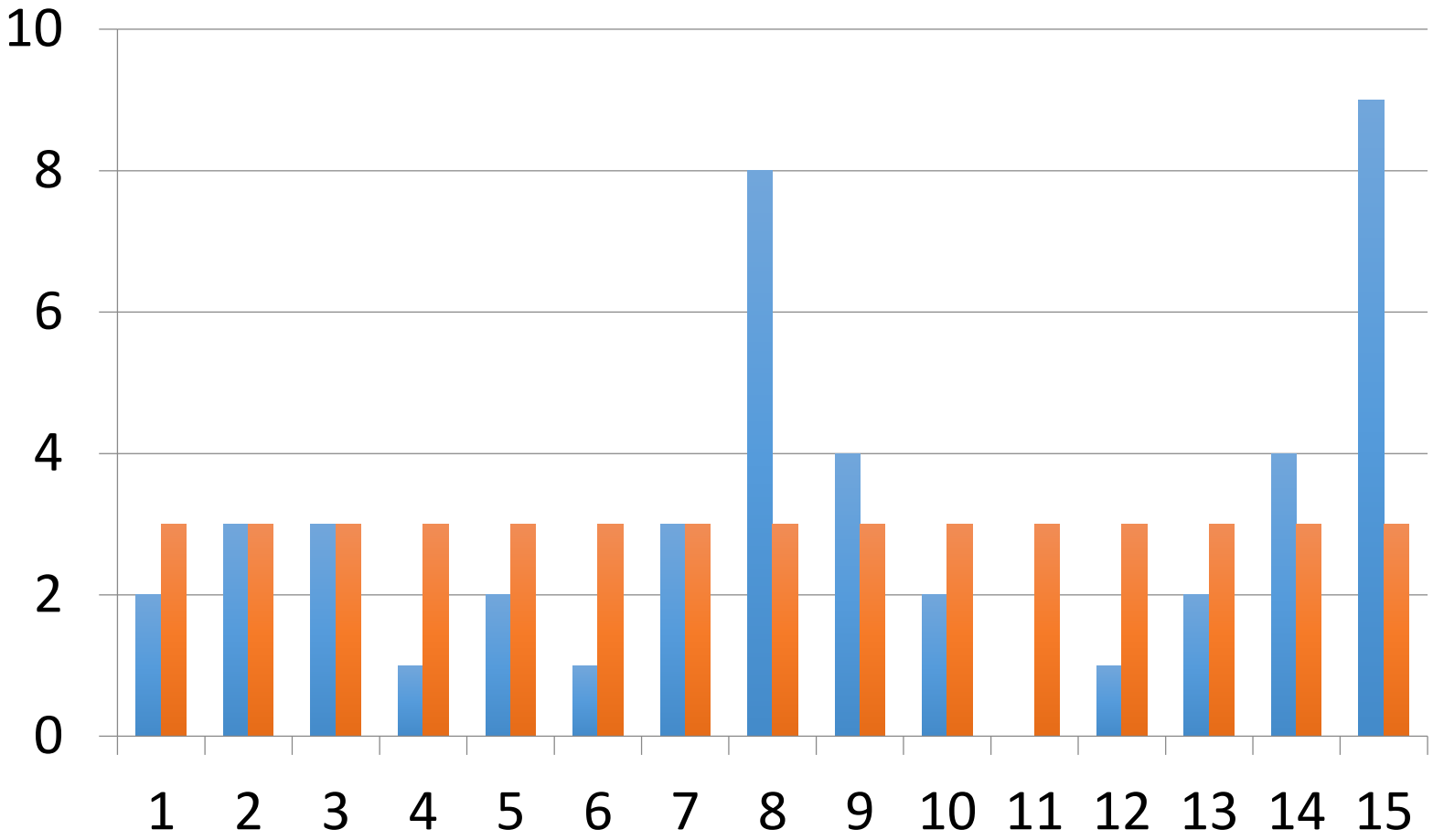
Example Histogram



How do we compute how many values between 8 and 10?
(Yes, it's obvious)

Problem: counts take up too much space!

Full vs. Uniform Counts



How much space do the full counts (bucket_size=1) take?

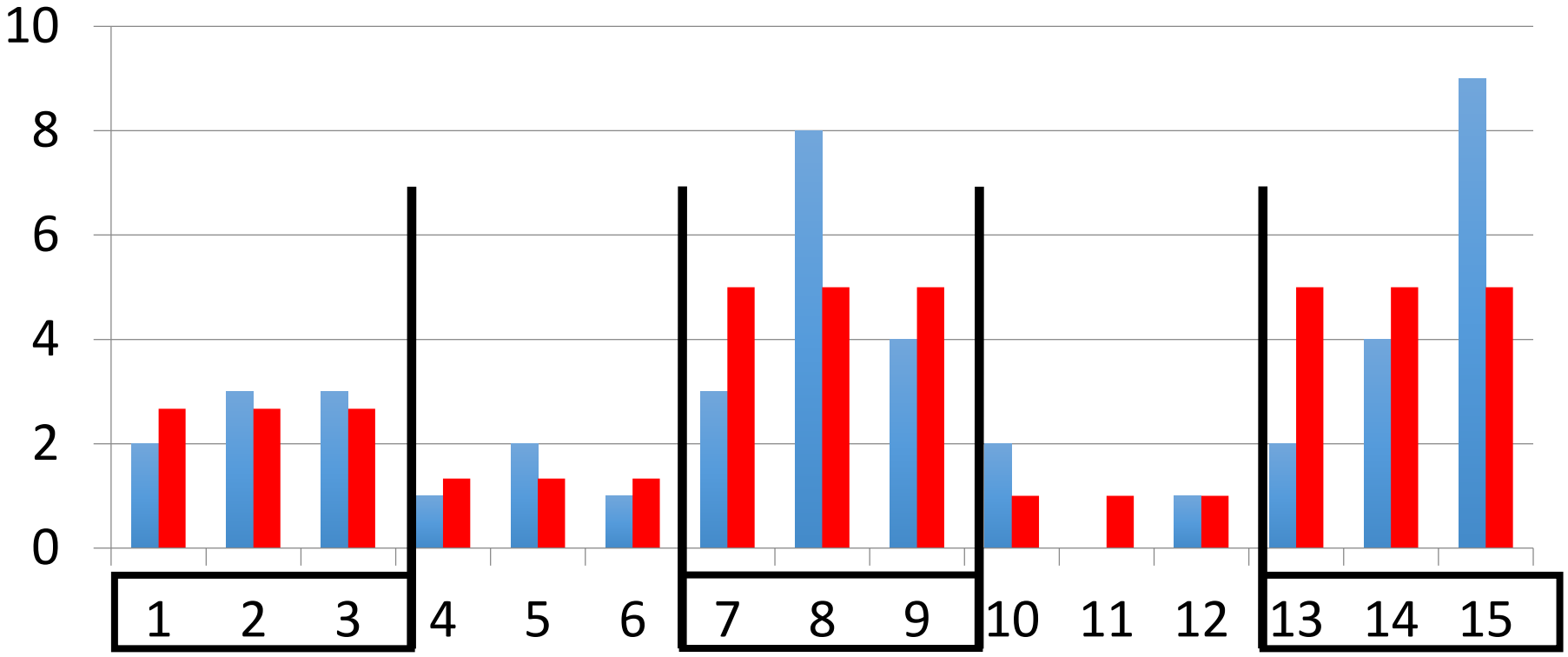
How much space do the uniform counts (bucket_size=ALL) take?

Fundamental Tradeoffs

- Want high resolution (like the full counts)
- Want low space (like uniform)
- Histograms are a compromise!

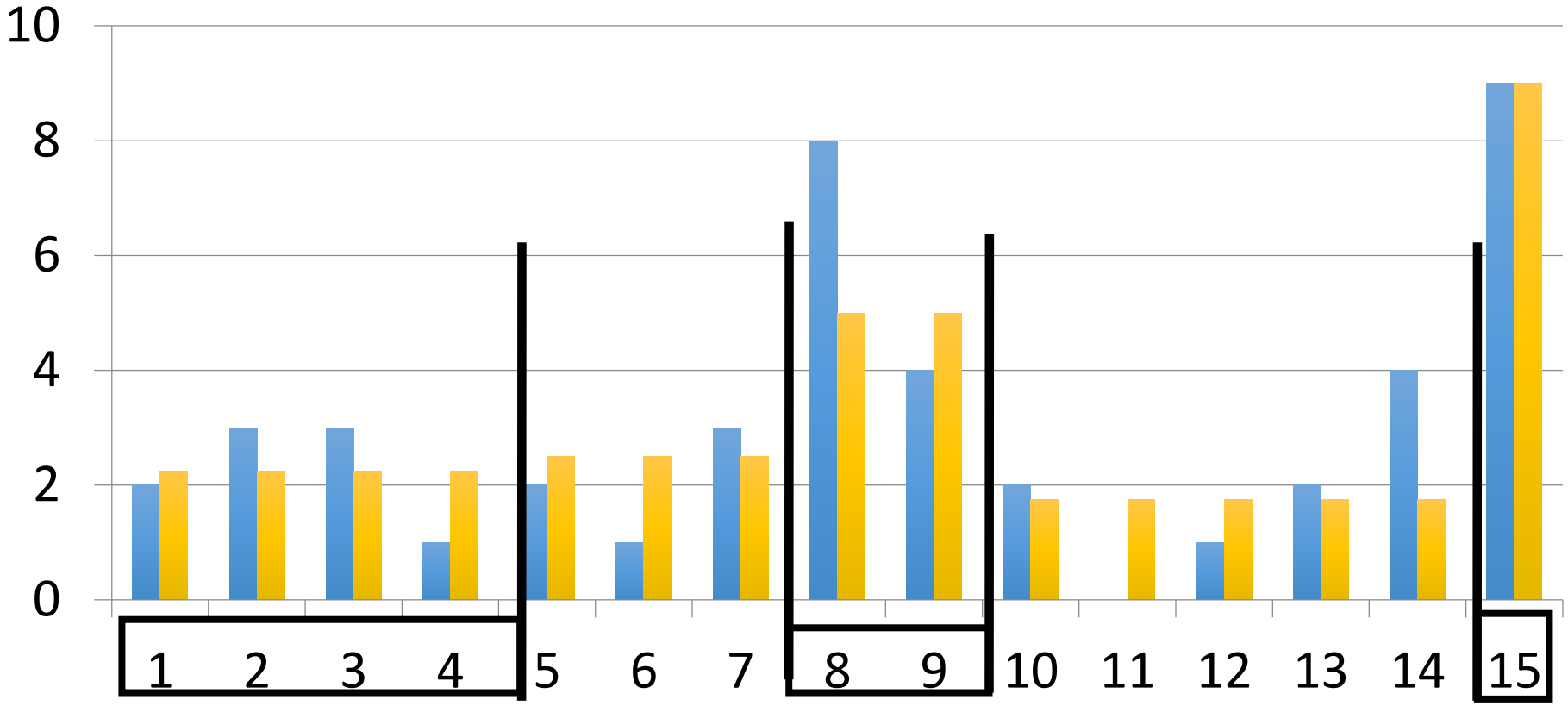
So how do we compute the “bucket” sizes?

Equi-width



All buckets roughly the same width

Equidepth



All buckets contain roughly the same number of items (total frequency). Able to adapt to skew.

Histograms

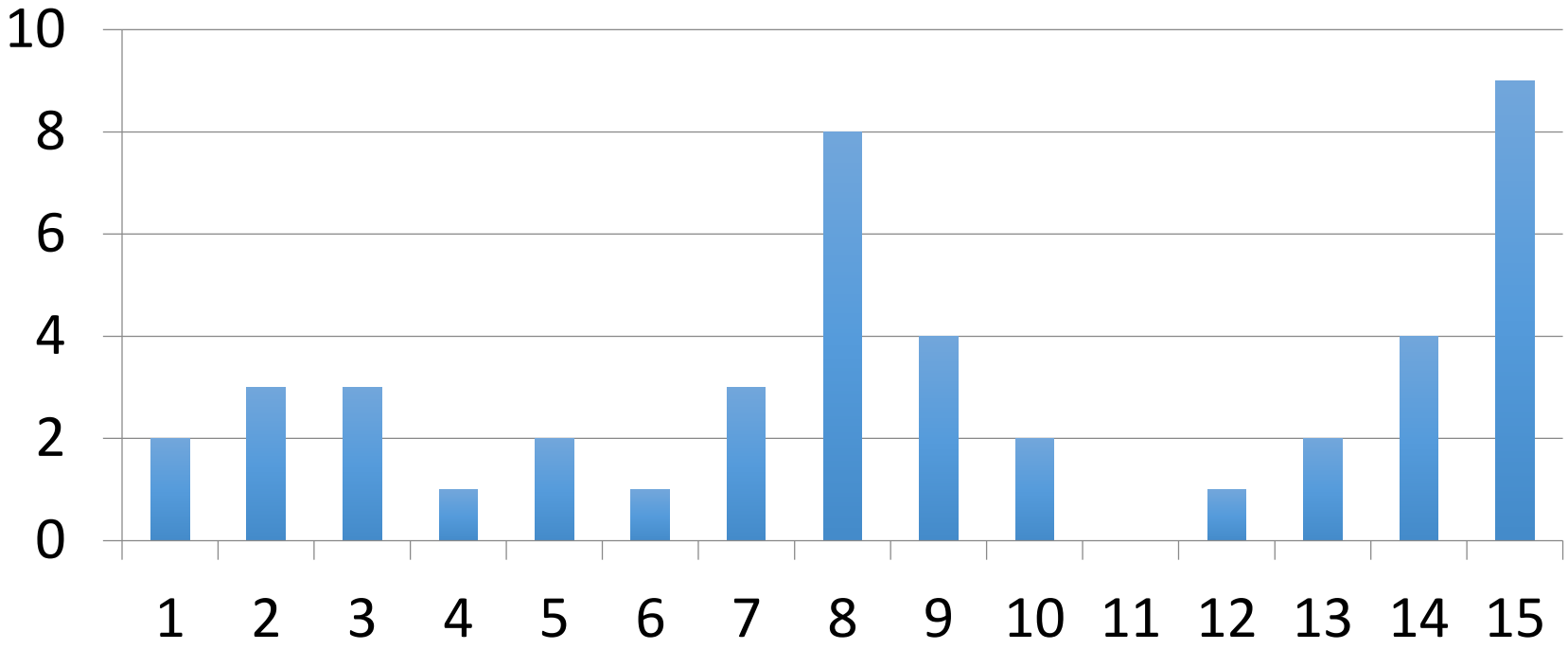
- Simple, intuitive and popular
- Parameters: # of buckets and type
- Can extend to many attributes (multidimensional)

Maintaining Histograms

- Histograms require that we update them!
  - Typically, you must run/schedule a command to update statistics on the database
  - Out of date histograms can be terrible!

- There is research work on self-tuning histograms and the use of query feedback
  - Oracle 11g

Nasty example



1. we insert many tuples with value > 16
2. we do **not** update the histogram
3. we ask for values > 20?

Precomputing Things

Views

- Views are relations, except that they are not physically stored.
- For presenting different information to different users
- Employee (ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
   SELECT name, project
   FROM  Employee
   WHERE department = "Development"
```

- Set privileges so that Payroll has access to Employee, others only to Developers

Example View Based on a Join

```
CREATE VIEW  Seattle-view  AS
      SELECT  buyer, seller, product, store
      FROM    Person, Purchase
      WHERE   Person.city = "Seattle"
                    AND Person.name = Purchase.buyer
```

We have a new virtual table:
Seattle-view (buyer, seller, product, store)

```
SELECT   name, store
FROM     Seattle-view, Product
WHERE    Seattle-view.product = Product.name
               AND Product.category = "shoes"
```

View is not Really a Table!

```
SELECT   name, Seattle-view.store
FROM      Seattle-view, Product
WHERE   Seattle-view.product = Product.name  AND
                  Product.category = "shoes"
```

This is what happens when you query a view

```
SELECT  name, Purchase.store
FROM   Person, Purchase, Product
WHERE  Person.city = "Seattle"    AND
                Person.name = Purchase.buyer  AND
                Purchase.product = Product.name  AND
                Product.category = "shoes"
```

Pros vs. Cons of Views

+ **Enforce Business Rules** – Use views to define business rules, such as when an items is active, or what is meant by “popular.”

+ **Consistency** – Simplify complicated query logic and calculations by hiding it behind the view’s definition.

+ **Security** – Restrict access to a table, yet allow users to access non-confidential data via views.

+ **Simplicity** – Databases with many tables possess complex relationships, which can be difficult to navigate if you aren’t comfortable using Joins.

− **Performance** – Each time a view is referenced, the query used to define it, is rerun.

− **Modifications** – Not all views support INSERT, UPDATE, or DELETE operations.

Materialized Views

- Unlike views, materialized views also store the results of the query in the database.

- Designed to improve the performance of the database by doing some intensive work in advance.
  - Can be used to pre-collect aggregate values
  - Assemble data that would come from many different tables, which would in turn require many different joins to be performed

Ordinary vs. Materialized Views

- Ordinary views
  - Virtual table
  - Named select statement
- Part of the SQL standard
- Syntax
  - CREATE VIEW *viewName* AS *selectStatement*

- Physical table
  - Replication of master data at a single point in time
- Not part of the SQL standard
- Syntax
  - CREATE **MATERIALIZED** VIEW *viewName* AS *selectStatement*

Why Use Materialized Views?

- Replicate data to non-master sites
  - To save network traffic when data is used in transactions
- Cache expensive queries
  - Expensive in terms of time or memory
  - Example: Sum, average or other calculations on large amounts of data

Types of Materialized Views

- Read-only
  - Insert, update or delete **NOT** allowed
- Updateable
  - Insert, update and delete on the view is allowed
  - Changes made to the view are pushed back to the master tables at refresh
- Writeable
  - Insert, update and delete on the view is allowed
  - Changes made to the view are **NOT** pushed back to the master tables at refresh

Refreshing a Materialized View

- Refresh types
  - Complete refresh
    - Recreates the materialized view
  - Fast (Incremental) refresh
    - Only changed data is refreshed
- Initiating a refresh
  - Scheduled refresh
  - On-demand refresh

Summary

- Database is doing lots of optimizations without you knowing
- Logical optimizations consider rewriting queries
- Physical optimizations uses additional structures to facilitate query evaluation

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the *CS145 Introduction to Databases* lecture given by *Christopher Ré* at *Stanford University* (http://web.stanford.edu/class/cs145/).

Chapter 13: Query Optimization

Database System Concepts, 6th Ed.

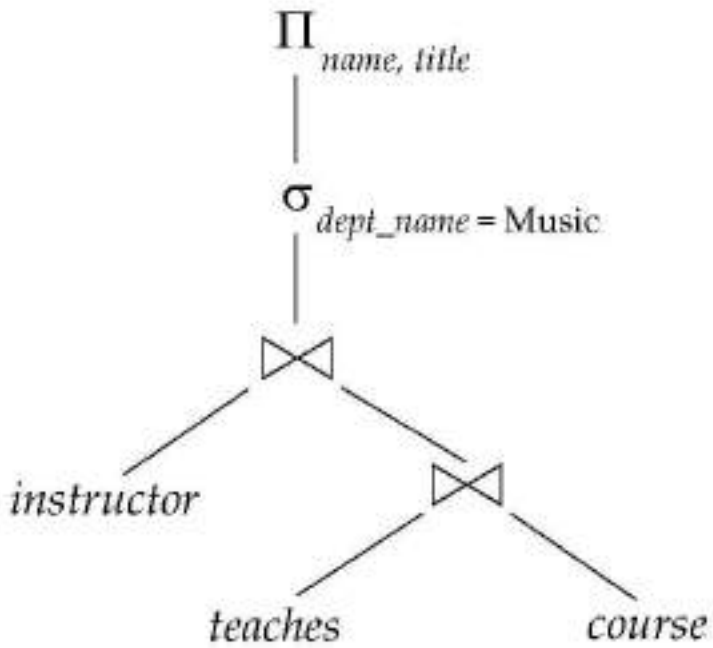
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

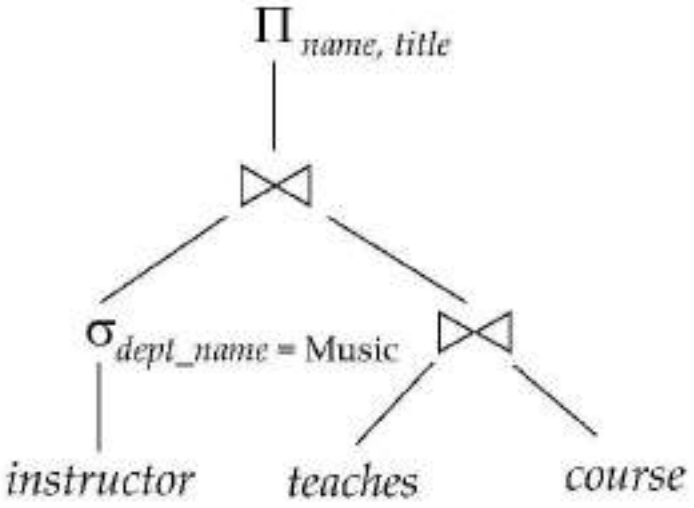
Chapter 13: Query Optimization

- Introduction
- Transformation of Relational Expressions
- Catalog Information for Cost Estimation
- Statistical Information for Cost Estimation
- Cost-based optimization
- Dynamic Programming for Choosing Evaluation Plans
- Materialized views

Introduction

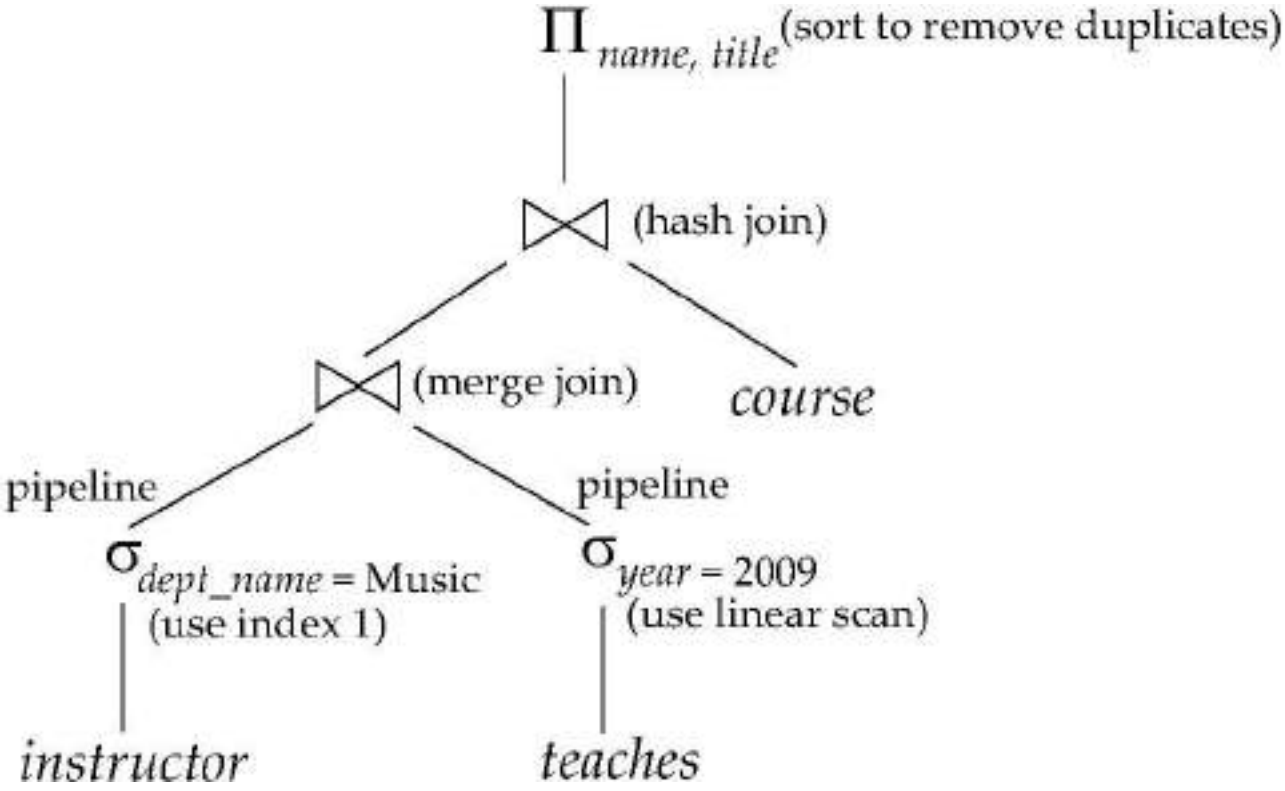
- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation





Introduction (Cont.)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



- Find out how to view query execution plans on your favorite database

Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Generating Equivalent Expressions

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every legal database instance
  - Note: order of tuples is irrelevant
  - we don't care if they generate different results on databases that violate integrity constraints
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{q_1 \wedge q_2}(E) = \sigma_{q_1}(\sigma_{q_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{q_1}(\sigma_{q_2}(E)) = \sigma_{q_2}(\sigma_{q_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural join operations are associative:

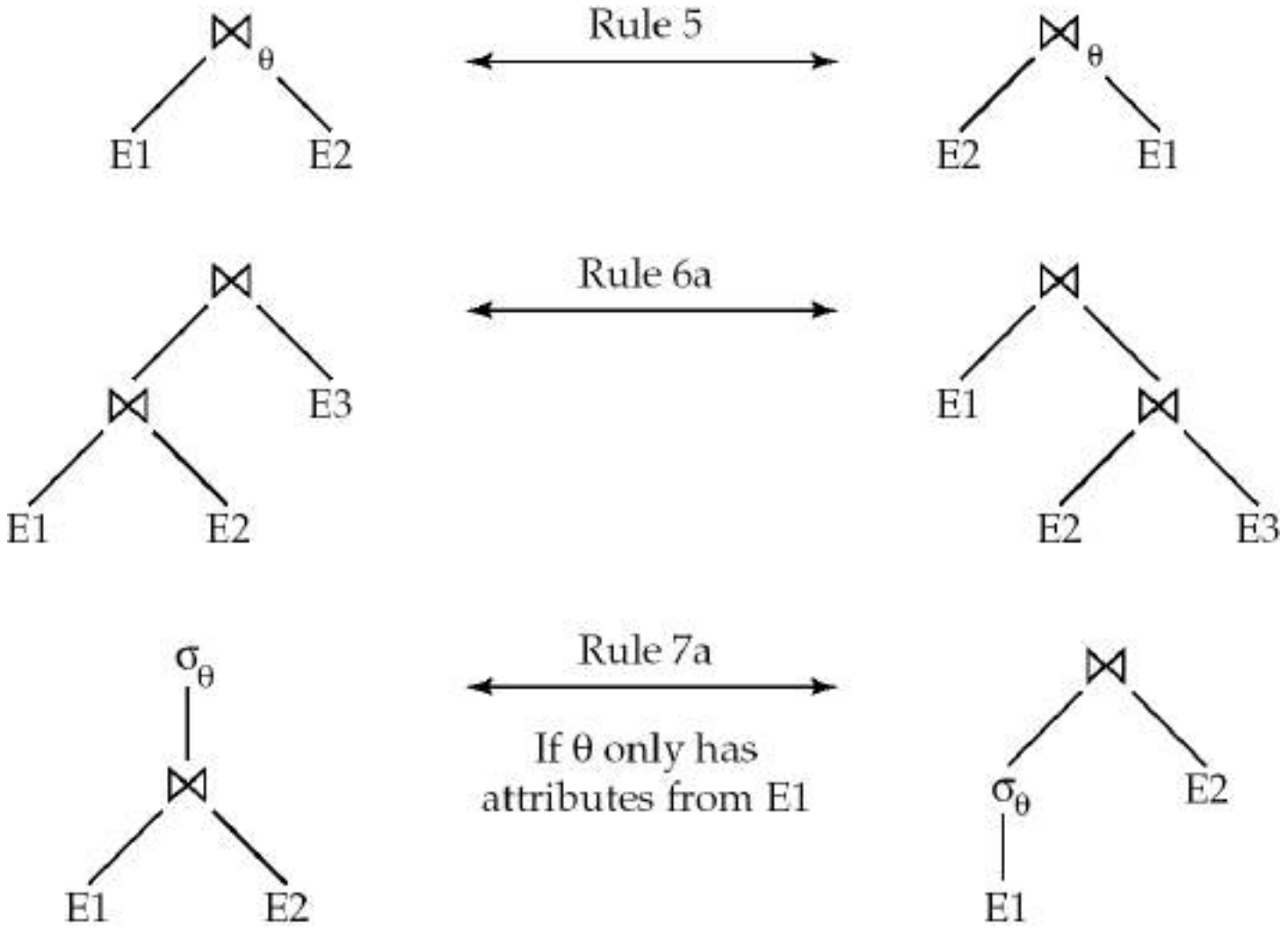
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$

Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

   (a) if θ involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   - (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and –.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of –

Also: $$\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

and similarly for $\cap$ in place of –, but not for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Exercise

- Create equivalence rules involving
  - The group by/aggregation operation
  - Left outer join operation

Transformation Example: Pushing Selections

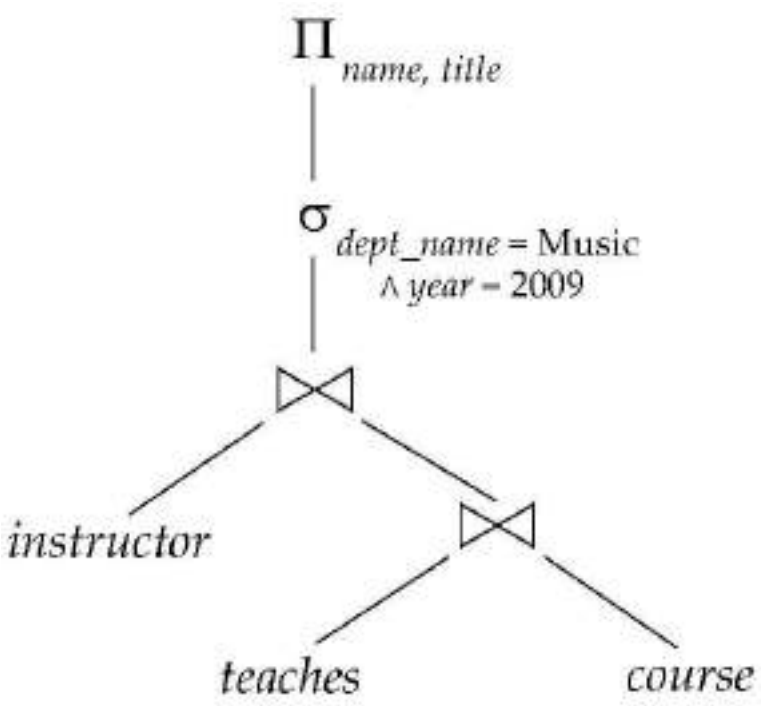
- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name,\ title}(\sigma_{dept_name=\ \text{“Music”}}$ $(instructor \bowtie (teaches \bowtie \Pi_{course_id,\ title}(course))))$
- Transformation using rule 7a.
  - $\Pi_{name,\ title}((\sigma_{dept_name=\ \text{“Music”}}(instructor)) \bowtie$ $(teaches \bowtie \Pi_{course_id,\ title}(course)))$
- Performing the selection as early as possible reduces the size of the relation to be joined.

Example with Multiple Transformations

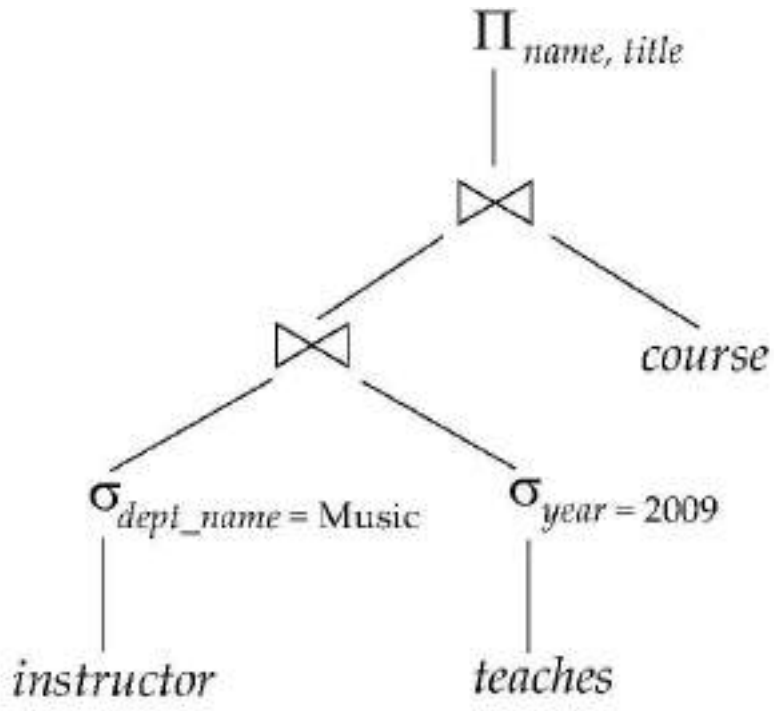
- Query: Find the names of all instructors in the Music department who have taught a course in 2009, along with the titles of the courses that they taught
  - $\Pi_{\text{name, title}}(\sigma_{\text{dept_name= "Music"} \wedge \text{year = 2009}}$ $(\text{instructor} \bowtie (\text{teaches} \bowtie \Pi_{\text{course_id, title}} (\text{course}))))$
- Transformation using join associatively (Rule 6a):
  - $\Pi_{\text{name, title}}(\sigma_{\text{dept_name= "Music"} \wedge \text{gear = 2009}}$ $((\text{instructor} \bowtie \text{teaches}) \bowtie \Pi_{\text{course_id, title}} (\text{course})))$
- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression

  $\sigma_{\text{dept_name = "Music"}} (\text{instructor}) \bowtie \sigma_{\text{year = 2009}} (\text{teaches})$

Multiple Transformations (Cont.)



(a) Initial expression tree



(b) Tree after multiple transformations

Transformation Example: Pushing Projections

- Consider: $\Pi_{name,\ title}(\sigma_{dept_name=\ \text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course_id,\ title}(course))))$
- When we compute

  $(\sigma_{dept_name\ =\ \text{“Music”}}(instructor) \bowtie teaches)$

  we obtain a relation whose schema is:
  (ID, name, dept_name, salary, course_id, sec_id, semester, year)
- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

  $\Pi_{name,\ title}(\Pi_{name,\ course_id}(\sigma_{dept_name=\ \text{“Music”}}(instructor) \bowtie teaches)) \bowtie \Pi_{course_id,\ title}(course))))$
- Performing the projection as early as possible reduces the size of the relation to be joined.

Join Ordering Example

- For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

  (Join Associativity)

- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

  so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{name,\ title}(\sigma_{dept_name=\ \text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course_id,\ title}(course))))$$

- Could compute teaches $\bowtie \Pi_{course_id,\ title}$ (course) first, and join result with

$$\sigma_{dept_name=\ \text{“Music”}}(instructor)$$

but the result of the first join is likely to be a large relation.

- Only a small fraction of the university's instructors are likely to be from the Music department
  - it is better to compute

$$\sigma_{dept_name=\ \text{“Music”}}(instructor) \bowtie teaches$$

first.

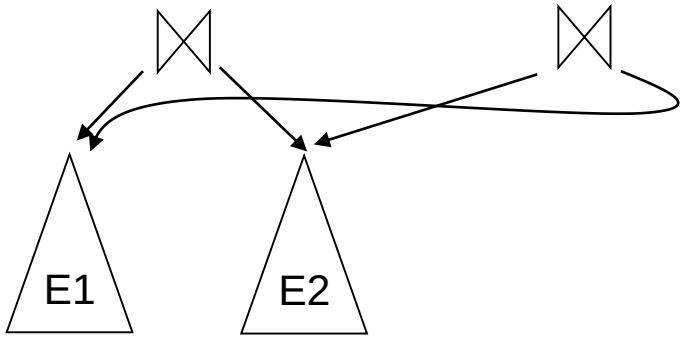
Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
  - Repeat
    - apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
    - add newly generated expressions to the set of equivalent expressions

    Until no new equivalent expressions are generated above
- The above approach is very expensive in space and time
  - Two approaches
    - Optimized plan generation based on transformation rules
    - Special case approach for queries with only selections, projections and joins

Implementing Transformation Based Optimization

- Space requirements reduced by sharing common sub-expressions:
  - when E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared using pointers
    - E.g. when applying join commutativity



  - Same sub-expression may get generated multiple times
    - Detect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
  - Dynamic programming
    - We will study only the special case of dynamic programming for join order optimization

Cost Estimation

- Cost of each operator computed as described in Chapter 12
  - Need statistics of input relations
    - E.g. number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g. number of distinct values for an attribute
- More on cost estimation later

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

Dynamic Programming in Optimization

- To find best join tree for a set of n relations:
  - To find best plan for a set S of n relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of S.
  - Recursively compute costs for joining subsets of S to find the cost of each plan. Choose the cheapest of the $2^n - 2$ alternatives.
  - Base case for recursion: single relation access plan
    - Apply all selections on R_i using best choice of indices on R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
    - Dynamic programming

Join Order Optimization Algorithm

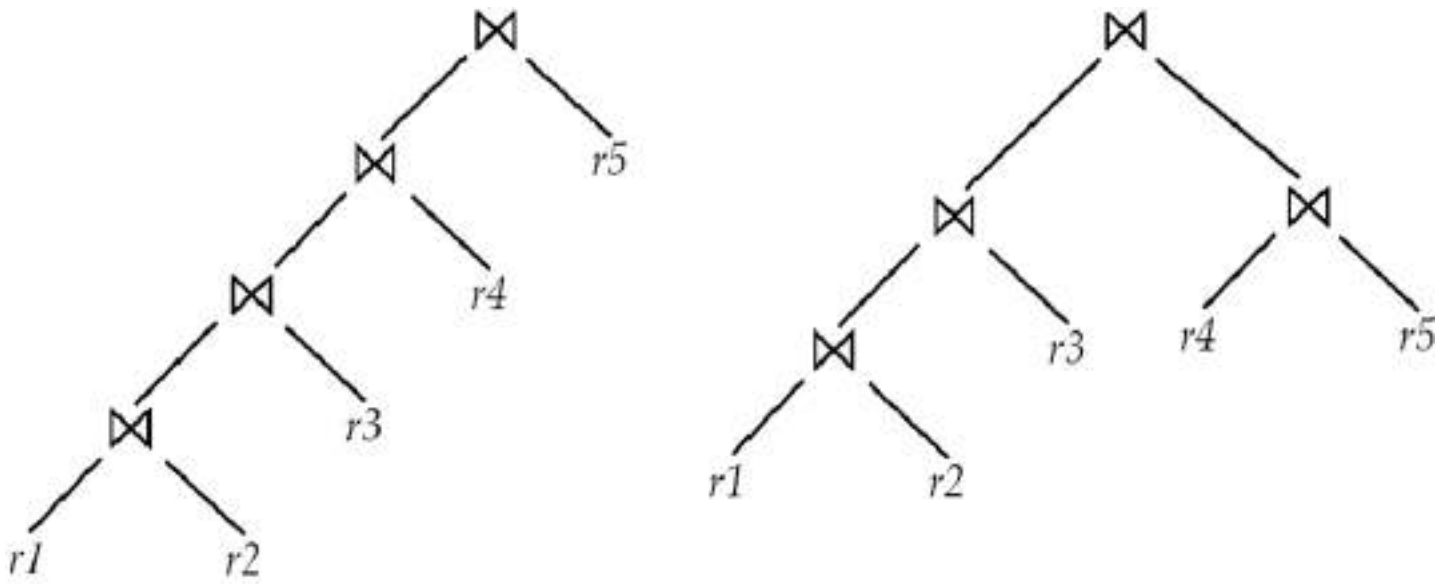
```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost based on the best way
        of accessing S  /* Using selections on S and indices on S */

    else for each non-empty subset S1 of S such that S1 ≠ S
        P1= findbestplan(S1)
        P2= findbestplan(S - S1)
        A = best algorithm for joining results of P1 and P2
        cost = P1.cost + P2.cost + cost of A
        if cost < bestplan[S].cost
            bestplan[S].cost = cost
            bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                join results of P1 and P2 using A"
    return bestplan[S]
```

* Some modifications to allow indexed nested loops joins on relations that have selections (see book)

Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



(a) Left-deep join tree

(b) Non-left-deep join tree

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Modify optimization algorithm:
    - Replace "**for each** non-empty subset S1 of S such that S1 $\neq$ S"
    - By: **for each** relation r in S
      let S1 = S – r .
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Interesting Sort Orders

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$ (with A as common attribute)
- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation
  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join but generates result sorted on A
  - Which in turn may make merge-join with r_3 cheaper, which may reduce cost of join with r_3 and minimizing overall cost
  - Sort order may also be useful for order by and for grouping
- Not sufficient to find the best join order for each subset of the set of n given relations
  - must find the best join order for each subset, **for each interesting sort order**
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

Cost Based Optimization with Equivalence Rules

- **Physical equivalence rules** allow logical query plan to be converted to physical query plan specifying what algorithms are used for each operation.
- Efficient optimizer based on equivalent rules depends on
  - A space efficient representation of expressions which avoids making multiple copies of subexpressions
  - Efficient techniques for detecting duplicate derivations of expressions
  - A form of dynamic programming based on **memoization**, which stores the best plan for a subexpression the first time it is optimized, and reuses in on repeated optimization calls on same subexpression
  - Cost-based pruning techniques that avoid generating all plans
- Pioneered by the Volcano project and implemented in the SQL Server optimizer

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Structure of Query Optimizers

- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimization
  - E.g. nested subqueries

Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
  - **Optimization cost budget** to stop optimization early (if cost of plan is less than cost of optimization)
  - **Plan caching** to reuse previously computed plan if query is resubmitted
    - Even with different constants in query
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

Statistics for Cost Estimation

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

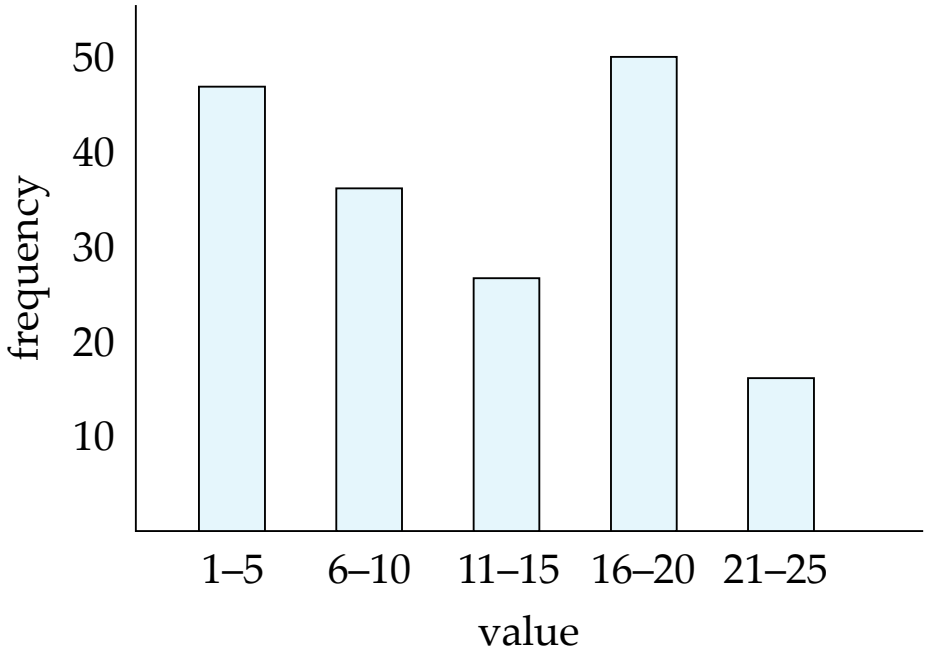
Statistical Information for Cost Estimation

- n_r: number of tuples in a relation r.
- b_r: number of blocks containing tuples of r.
- l_r: size of a tuple of r.
- f_r: blocking factor of r — i.e., the number of tuples of r that fit into one block.
- V(A, r): number of distinct values that appear in r for attribute A; same as the size of $\Pi_A(r)$.
- If tuples of r are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Histograms

- Histogram on attribute age of relation person



- **Equi-width** histograms
- **Equi-depth** histograms

Selection Size Estimation

- $\sigma_{A=v}(r)$
  - n_r / V(A,r) : number of records that will satisfy the selection
  - Equality condition on a key attribute: size estimate = 1
- $\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying the condition.
  - If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r)
    - c = $n_r . \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information c is assumed to be n_r / 2.

Size Estimation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation r satisfies θ_i .
  - If s_i is the number of satisfying tuples in r, the selectivity of θ_i is given by s_i / n_r.
- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. Assuming indepdence, estimate of tuples in the result is: $n_r * \dfrac{s_1 * s_2 * \ldots * s_n}{n_r^n}$
- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples:

  $n_r - \text{size}(\sigma_\theta(r))$

Join Operation: Running Example

Running example:

student $\bowtie$ takes

Catalog information for join examples:

- $n_{student}$ = 5,000.
- $f_{student}$ = 50, which implies that
  $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000.
- f_{takes} = 25, which implies that
  b_{takes} = 10000/25 = 400.
- V(ID, takes) = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute ID in takes is a foreign key referencing student.
  - V(ID, student) = 5000 (primary key!)

Estimation of the Size of Joins

- The Cartesian product r x s contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \varnothing$, then $r \bowtie s$ is the same as r x s.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s.
- If $R \cap S$ in S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
  - The case for $R \cap S$ being a foreign key referencing S is symmetric.
- In the example query student $\bowtie$ takes, ID in takes is a foreign key referencing student
  - hence, the result has exactly n_{takes} tuples, which is 10000

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple t in R produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.
- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for depositor $\bowtie$customer without using information about foreign keys:
  - V(ID, takes) = 2500, and
    V(ID, student) = 5000
  - The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/5000 = 10000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\Pi_A(r)$ = V(A,r)
- Aggregation : estimated size of ${}_A\mathcal{G}_F(r)$ = V(A,r)
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(r)$
  - For operations on different relations:
    - estimated size of $r \cup s$ = size of r + size of s.
    - estimated size of $r \cap s$ = minimum size of r and size of s.
    - estimated size of $r - s$ = r.
    - <u>All the three estimates may be quite inaccurate, but provide upper bounds on the sizes</u>.

Size Estimation (Cont.)

- Outer join:
  - Estimated size of r ⟕ s = size of r ⋈ s + size of r
    - Case of right outer join is symmetric
  - Estimated size of r ⟗ s = size of r ⋈ s + size of r + size of s

Estimation of Number of Distinct Values

Selections: σ_θ (r)

- If θ forces A to take a specified value: $V(A,\sigma_\theta(r)) = 1$.
  - e.g., A = 3
- If θ forces A to take on one of a specified set of values:
  $V(A,\sigma_\theta(r))$ = number of specified values.
  - (e.g., (A = 1 V A = 3 V A = 4)),
- If the selection condition θ is of the form A op r
  estimated $V(A,\sigma_\theta(r)) = V(A.r) * s$
  - where s is the selectivity of the selection.
- In all the other cases: use approximate estimate of
  $\min(V(A,r), n_{\sigma\theta(r)})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

Joins: $r \bowtie s$

- If all attributes in A are from r
  estimated $V(A, r \bowtie s) = \min(V(A,r), n_{r \bowtie s})$
- If A contains attributes A1 from r and A2 from s, then estimated $V(A, r \bowtie s) =$

  $$\min(V(A1,r)*V(A2 - A1,s), V(A1 - A2,r)*V(A2,s), n_{r \bowtie s})$$

  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in $\Pi_{A\ (r)}$ as in r.
- The same holds for grouping attributes of aggregation.
- For aggregated values
  - For min(A) and max(A), the number of distinct values can be estimated as min(V(A,r), V(G,r)) where G denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use V(G,r)

Additional Optimization Techniques

- Nested Subqueries
- Materialized Views

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Optimizing Nested Subqueries**

- Nested query example:
  select name
  from instructor
  where exists (**select** *
  from teaches
  where instructor.ID = teaches.ID **and** teaches.year = 2007)
- SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**
- Conceptually, nested subquery is executed once for each tuple in the cross-product generated by the outer level **from** clause
  - Such evaluation is called **correlated evaluation**
  - Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery

Optimizing Nested Subqueries (Cont.)

- Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - there may be unnecessary random I/O as a result
- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques
- E.g.: earlier nested query can be rewritten as
  select name
  from instructor, teaches
  where instructor.ID = teaches.ID **and** teaches.year = 2007
  - Note: the two queries generate different numbers of duplicates (why?)
    - teaches can have duplicate IDs
    - Can be modified to handle duplicates correctly as we will see
- In general, it is not possible/straightforward to move the entire nested subquery from clause into the outer level query from clause
  - A temporary relation is created instead, and used in body of outer level query

Optimizing Nested Subqueries (Cont.)

In general, SQL queries of the form below can be rewritten as shown

- Rewrite: **select** …
  from L_1
  where P_1 **and exists** (**select** *
  from L_2
  where P_2)
- To: **create table** t_1 as
  select distinct V
  from L_2
  where P_2^1

  select …
  from L_1, t_1
  where P_1 **and** P_2^2
  - P_2^1 contains predicates in P_2 that do not involve any correlation variables
  - P_2^2 reintroduces predicates involving correlation variables, with relations renamed appropriately
  - V contains all attributes used in predicates with correlation variables

Optimizing Nested Subqueries (Cont.)

- In our example, the original nested query would be transformed to

```
create table t1 as
    select distinct ID
    from teaches
    where year = 2007

select name
from instructor, t1
 where t1.ID = instructor.ID
```

- The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.
- Decorrelation is more complicated when
  - the nested subquery uses aggregation, or
  - when the result of the nested subquery is used to test for equality, or
  - when the condition linking the nested subquery to the other query is **not exists**,
  - and so on.

Materialized Views**

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view
  c**reate view** department_total_salary(dept_name, total_salary) **as**
  select dept_name, **sum**(salary)
  from instructor
  group by dept_name
- Materializing the above view would be very useful if the total salary by department is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A better option is to use **incremental view maintenance**
  - **Changes to database relations are used to compute changes to the materialized view, which is then updated**
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic recomputation (e.g. nightly)
  - Above methods are directly supported by many database systems
    - Avoids manual effort/correctness issues

Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from r are denoted $\mathbf{i_r}$ and $\mathbf{d_r}$
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple
- We describe how to compute the change to the result of each relational operation, given changes to its inputs
- We then outline how to handle relational algebra expressions

Join Operation

- Consider the materialized view $v = r \bowtie s$ and an update to r
- Let r^{old} and r^{new} denote the old and new states of relation r
- Consider the case of an insert to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{old} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
- Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$

A, 1
B, 2

C,2

$\bowtie$

1, p
2, r
2, s

A, 1, p
B, 2, r
B, 2, s

C, 2, r
C, 2, s

Selection and Projection Operations

- Selection: Consider a view $v = \sigma_\theta(r)$.
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$
- Projection is a more difficult operation
  - R = (A,B), and r(R) = { (a,2), (a,3)}
  - $\Pi_A(r)$ has a single tuple (a).
  - If we delete the tuple (a,2) from r, we should not delete the tuple (a) from $\Pi_A(r)$, but if we then delete (a,3) as well, we should delete the tuple
- For each tuple in a projection $\Pi_A(r)$, we will keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\Pi_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\Pi_A(r)$
    - if the count becomes 0, we delete the tuple from $\Pi_A(r)$

Aggregation Operations

- count : v = ${}_{A}\mathcal{g}_{count(B)}{}^{(r)}$.
  - When a set of tuples i_r is inserted
    - For each tuple r in i_r, if the corresponding group is already present in v, we increment its count, else we add a new tuple with count = 1
  - When a set of tuples d_r is deleted
    - for each tuple t in i_r. we look for the group t.A in v, and subtract 1 from the count for the group.
      - If the count becomes 0, we delete from v the tuple for the group t.A
- sum: v = ${}_{A}\mathcal{g}_{sum\ (B)}{}^{(r)}$
  - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
  - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from v
    - Cannot simply test for sum = 0 (why?)
- To handle the case of **avg**, we maintain the **sum** and **count** aggregate values separately, and divide at the end

Aggregate Operations (Cont.)

- **min**, **max**: $v = {}_{A}\mathcal{g}_{\min(B)}(r)$.
  - Handling insertions on r is straightforward.
  - Maintaining the aggregate values **min** and **max** on deletions may be more expensive. We have to look at the other tuples of r that are in the same group to find the new minimum

Other Operations

- Set intersection: $v = r \cap s$
  - when a tuple is inserted in r we check if it is present in s, and if so we add it to v.
  - If the tuple is deleted from r, we delete it from the intersection if it is present.
  - Updates to s are symmetric
  - The other set operations, union and set difference are handled in a similar fashion.
- Outer joins are handled in much the same way as joins but with some extra work
  - we leave details to you.

Handling Expressions

- To handle an entire expression, we derive expressions for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.
- E.g. consider $E_1 \bowtie E_2$ where each of E_1 and E_2 may be a complex expression
  - Suppose the set of tuples to be inserted into E_1 is given by D_1
    - Computed earlier, since smaller sub-expressions are handled first
  - Then the set of tuples to be inserted into $E_1 \bowtie E_2$ is given by $D_1 \bowtie E_2$
    - This is just the usual way of maintaining joins

Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$
  - We can rewrite the query as $v \bowtie t$
    - Whether to do so depends on cost estimates for the two alternative
- Replacing a use of a materialized view by the view definition:
  - A materialized view $v = r \bowtie s$ is available, but without any index on it
  - User submits a query $\sigma_{A=10}(v)$.
  - Suppose also that s has an index on the common attribute B, and r has an index on attribute A.
  - The best plan for this query may be to replace v by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$
- Query optimizer should be extended to consider all above alternatives and choose the best overall plan

Materialized View Selection

- **Materialized view selection**: "What is the best set of views to materialize?".
- **Index selection:** "what is the best set of indices to create"
  - closely related, to materialized view selection
    - but simpler
- Materialized view selection and index selection based on typical system **workload** (queries and updates)
  - Typical goal: minimize time to execute workload , subject to constraints on space and time taken for some critical queries/updates
  - One of the steps in database tuning
    - more on tuning in later chapters
- Commercial database systems provide tools (called "tuning assistants" or "wizards") to help the database administrator choose what indices and materialized views to create

Additional Optimization Techniques

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Top-K Queries

- **Top-K queries**

```
select *
from r, s
where r.B = s.B
order by r.A ascending
limit 10
```

  - Alternative 1: Indexed nested loops join with r as outer
  - Alternative 2: estimate highest r.A value in result and add selection (**and** r.A <= H) to where clause
    - If < 10 results, retry with larger H

Optimization of Updates

- **Halloween problem**

  update R **set** A = 5 * A
  where A > 10

  - If index on A is used to find tuples satisfying A > 10, and tuples updated immediately, same tuple may be found (and updated) multiple times
  - Solution 1: Always defer updates
    - collect the updates (old and new values of tuples) and update relation and indices in second pass
    - Drawback: extra overhead even if e.g. update is only on R.B, not on attributes in selection condition
  - Solution 2: Defer only if required
    - Perform immediate update if update does not affect attributes in where clause, and deferred updates otherwise.

Join Minimization

- **Join minimization**

  select r.A, r.B
  from r, s
  where r.B = s.B

- Check if join with s is redundant, drop it
  - E.g. join condition is on foreign key from r to s, r.B is declared as not null, and no selection on s
  - Other sufficient conditions possible

    select r.A, s2.B
    from r, s **as** s1, s **as** s2
    where r.B=s1.B **and** r.B = s2.B **and** s1.A < 20 **and** s2.A < 10

    - join with s1 is redundant and can be dropped (along with selection on s1)
  - Lots of research in this area since 70s/80s!

Multiquery Optimization

- Example

  Q1: **select** * **from** (r **natural join** t) **natural join** s

  Q2: **select** * **from** (r **natural join** u) **natural join** s

  - Both queries share common subexpression (r natural join s)
  - May be useful to compute (r natural join s) once and use it in both queries
    - But this may be more expensive in some situations
      - e.g. (r natural join s) may be expensive, plans as shown in queries may be cheaper
- **Multiquery optimization**: find best overall plan for a set of queries, expoiting sharing of common subexpressions between queries where it is useful

Multiquery Optimization (Cont.)

- Simple heuristic used in some database systems:
  - optimize each query separately
  - detect and exploiting common subexpressions in the individual optimal query plans
    - May not always give best plan, but is cheap to implement
  - **Shared scans**: widely used special case of multiquery optimization
- Set of materialized views may share common subexpressions
  - As a result, view maintenance plans may share subexpressions
  - Multiquery optimization can be useful in such situations

Parametric Query Optimization

- Example
  select *
  from r **natural join** s
  where r.a < $1
  - value of parameter $1 not known at compile time
    - known only at run time
  - different plans may be optimal for different values of $1
- Solution 1: optimize at run time, each time query is submitted
    - can be expensive
- Solution 2: **Parametric Query Optimization**:
  - optimizer generates a set of plans, optimal for different values of $1
    - Set of optimal plans usually small for 1 to 3 parameters
    - Key issue: how to do find set of optimal plans efficiently
  - best one from this set is chosen at run time when $1 is known
- Solution 3: **Query Plan Caching**
  - If optimizer decides that same plan is likely to be optimal for all parameter values, it caches plan and reuses it, else reoptimize each time
  - Implemented in many database systems

Extra Slides

(Not in 6th Edition book)

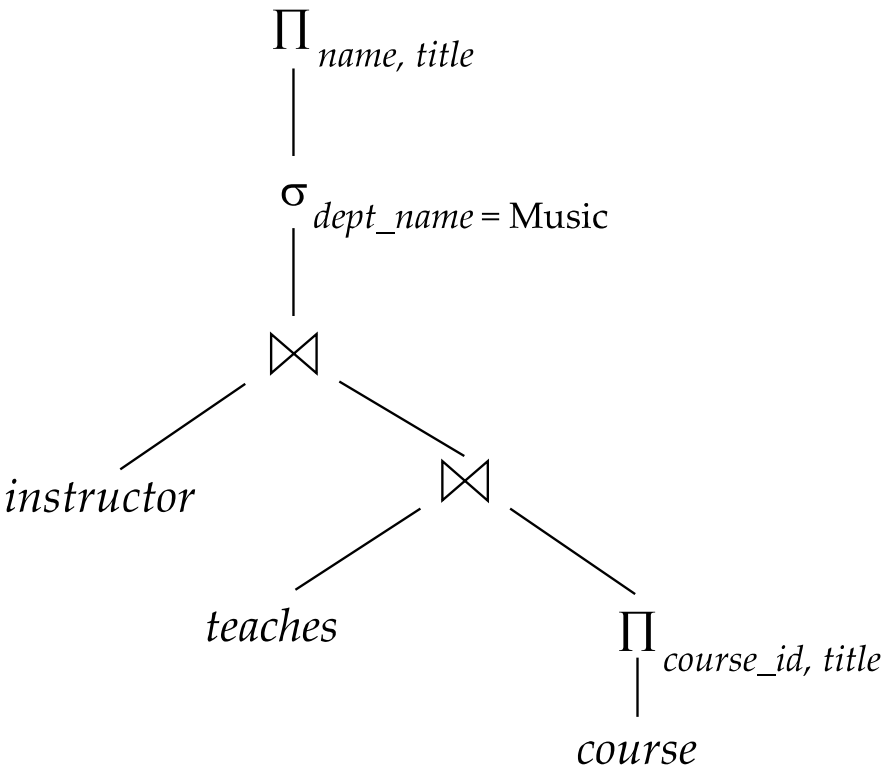
Plan Stability Across Optimizer Changes

- What if 95% of plans are faster on database/optimizer version N+1 than on N, but 5% are slower?
  - Why should plans be slower on new improved optimizer?
    - Answer: Two wrongs can make a right, fixing one wrong can make things worse!
- Approaches:
  - Allow hints for tuning queries
    - Not practical for migrating large systems with no access to source code
  - Set optimization level, default to version N (Oracle)
    - And migrate one query at a time after testing both plans on new optimizer
  - Save plan from version N, and give it to optimizer version N+1
    - Sybase, XML representation of plans (SQL Server)

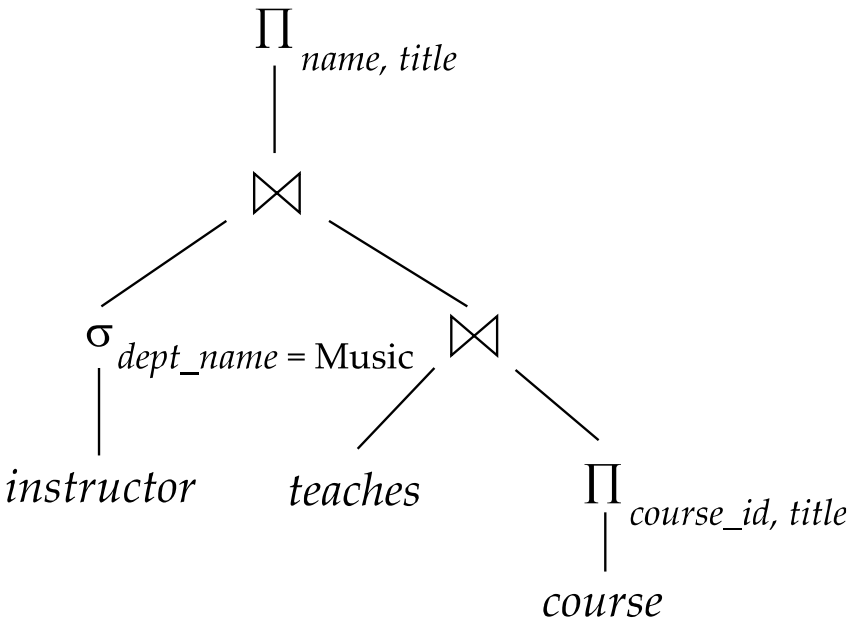
End of Chapter

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Figure 13.01



(a) Initial expression tree



(b) Transformed expression tree

Figure 13.02

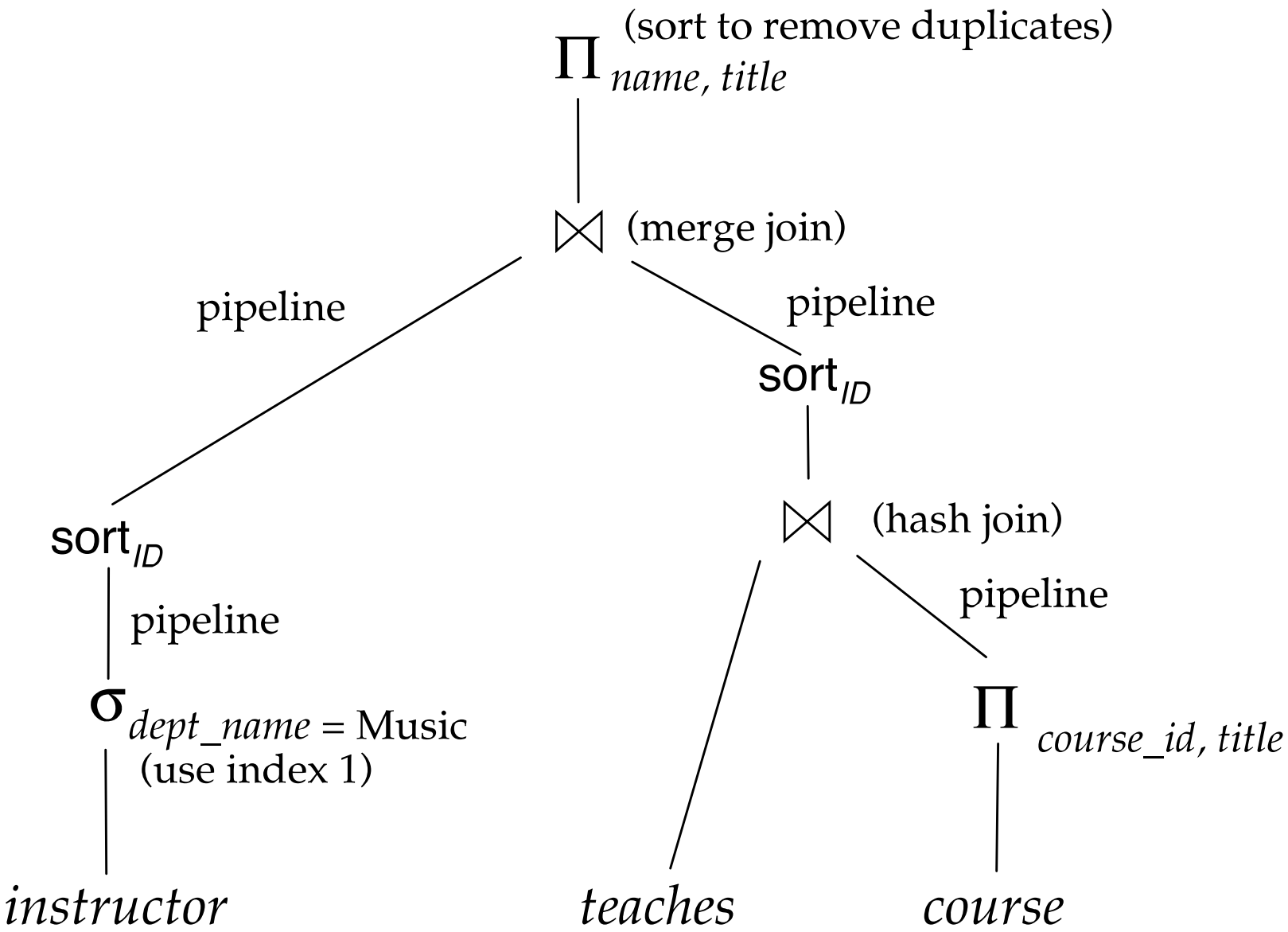

(sort to remove duplicates)
Π name, title
⋈ (merge join)
pipeline
pipeline
sort ID
sort ID
⋈ (hash join)
pipeline
pipeline
σ dept_name = Music
(use index 1)
Π course_id, title
instructor
teaches
course

Figure 13.03

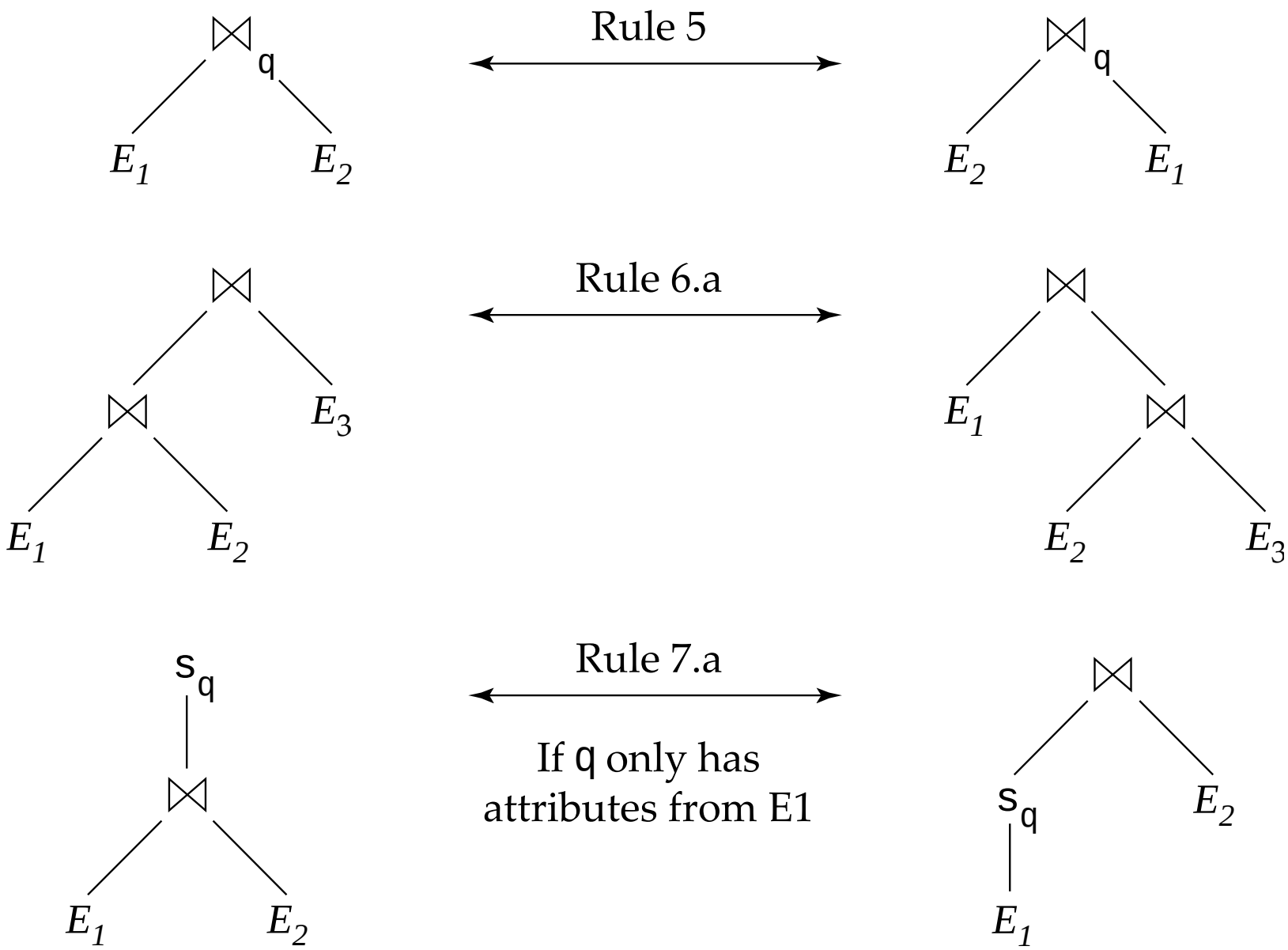
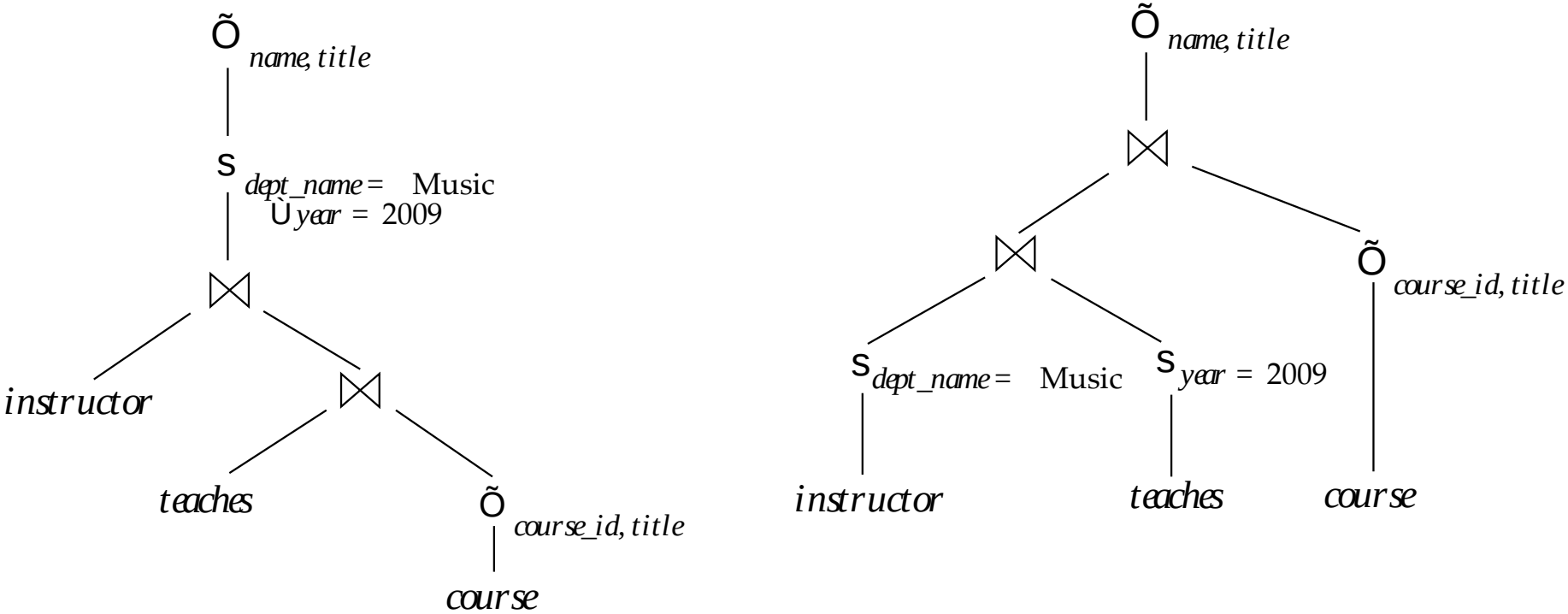

Rule 5
q
E_1
E_2
q
E_2
E_1
Rule 6.a
E_3
E_1
E_2
E_1
E_2
E_3
Rule 7.a
s_q
E_1
E_2
If q only has
attributes from E1
s_q
E_2
E_1

Figure 13.04



(a) Initial expression tree

(b) Tree after multiple transformations

Figure 13.06

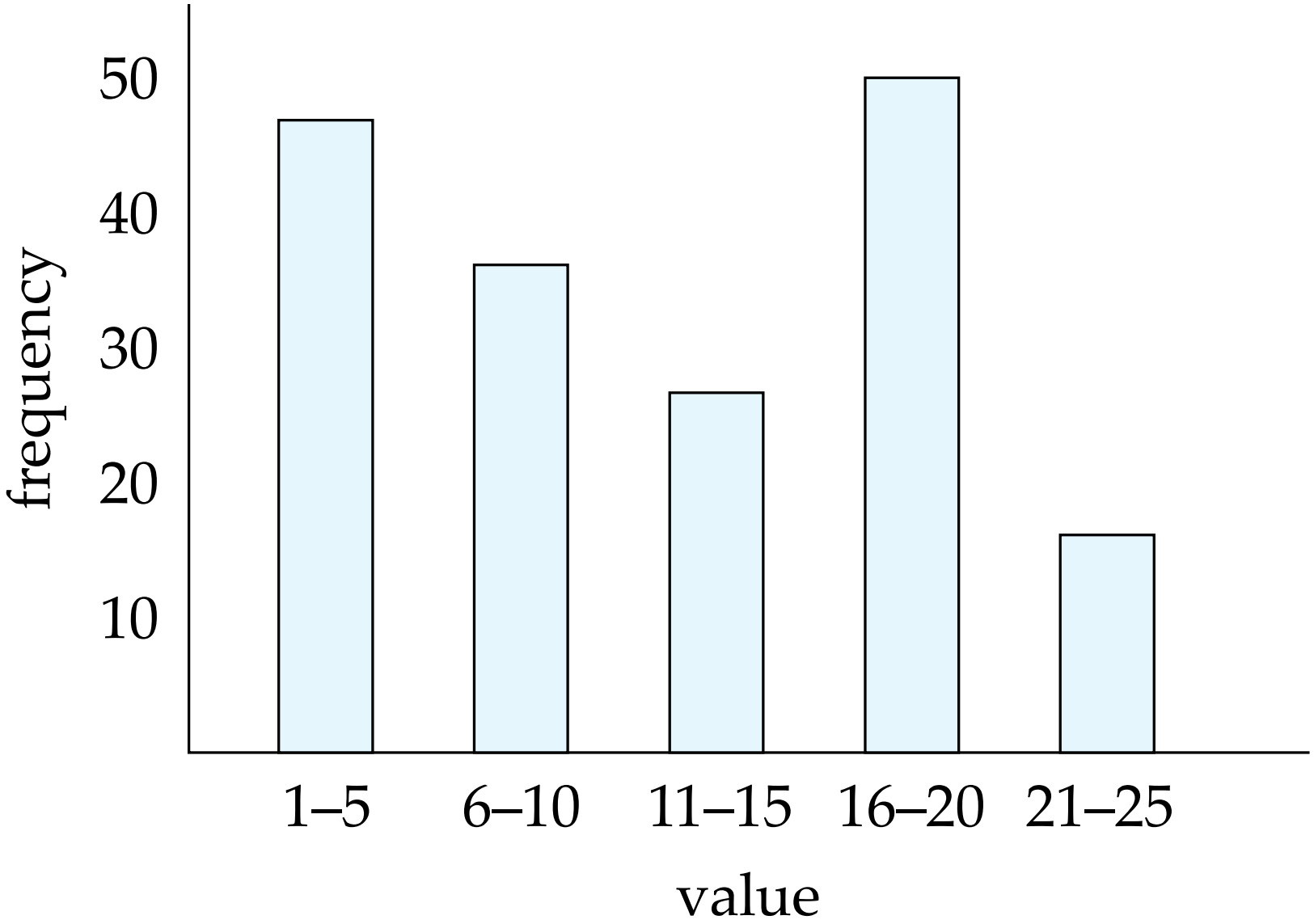
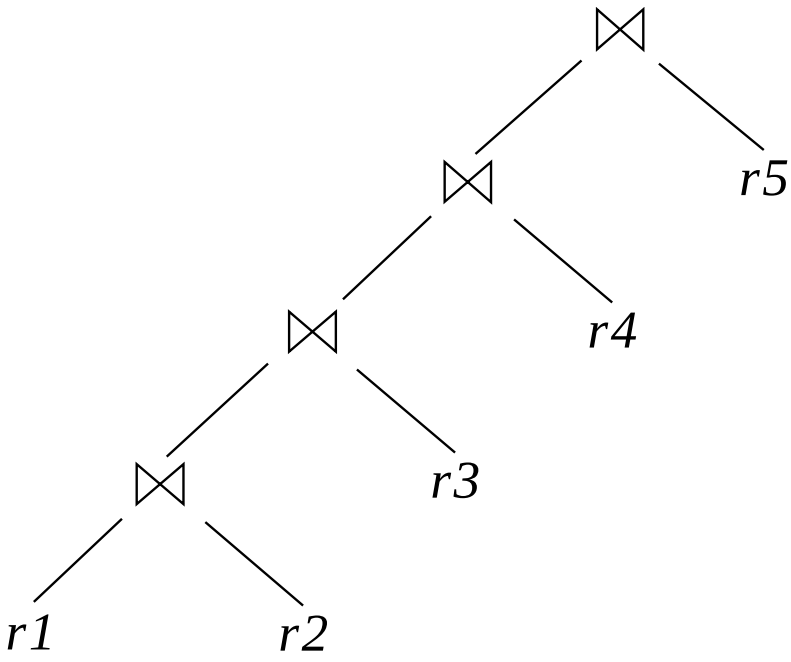
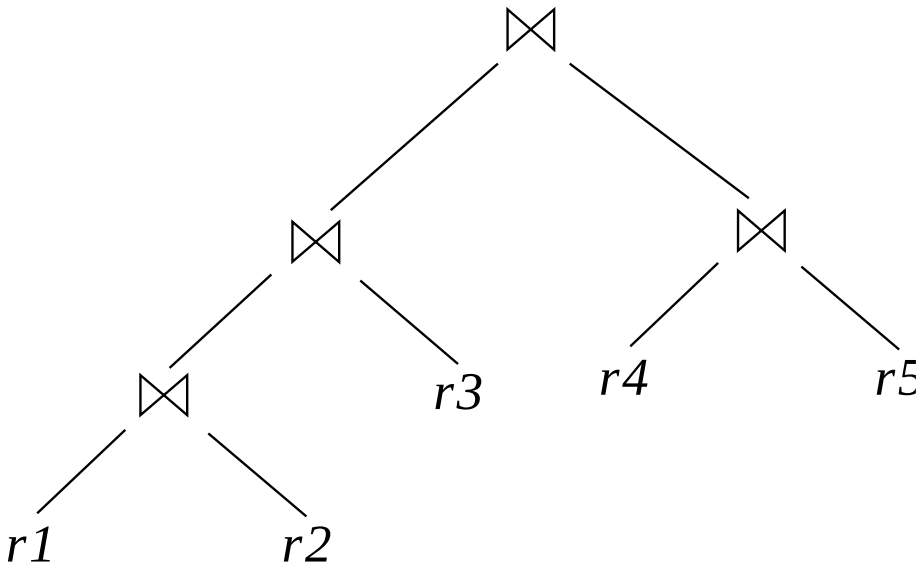


Figure 13.08



(a) Left-deep join tree



(b) Non-left-deep join tree