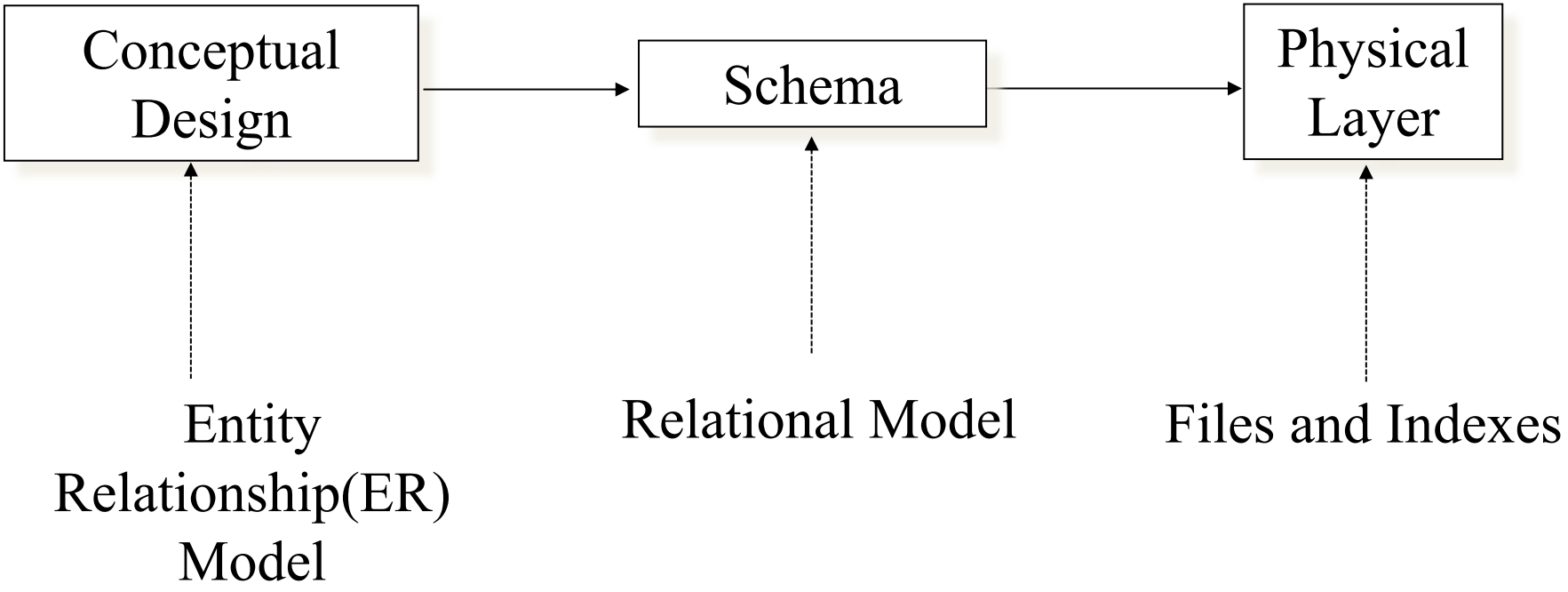
CS 540
Database Management Systems

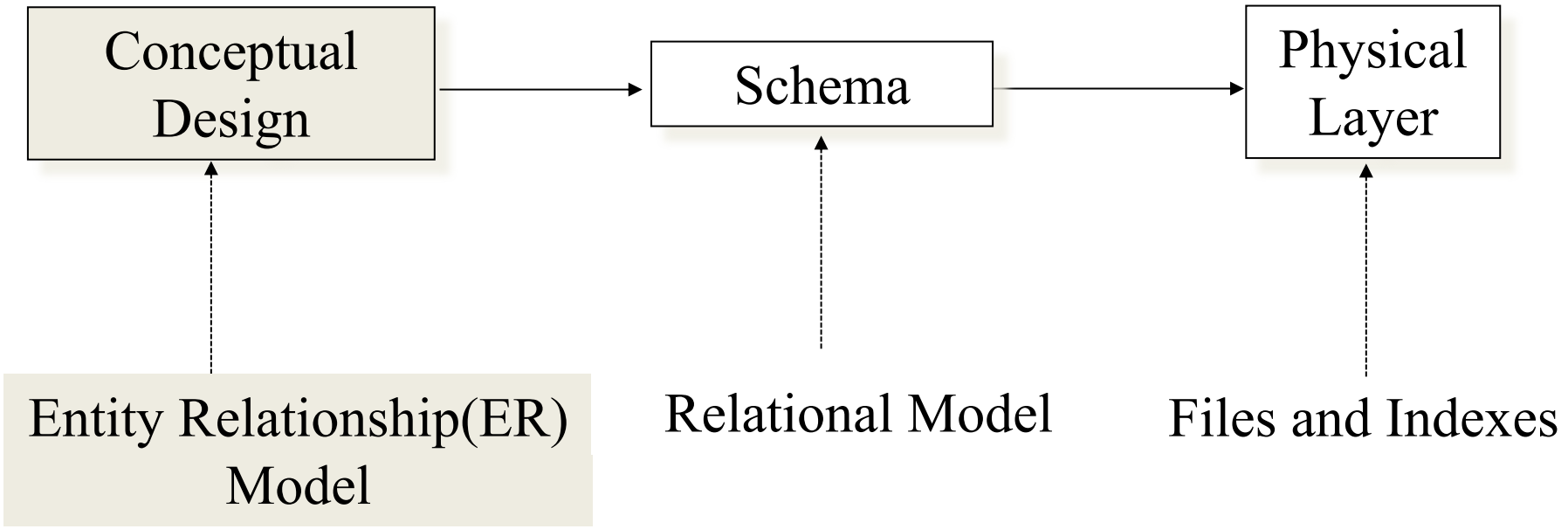
Review of Relational Model and Query Languages

Oregon State University

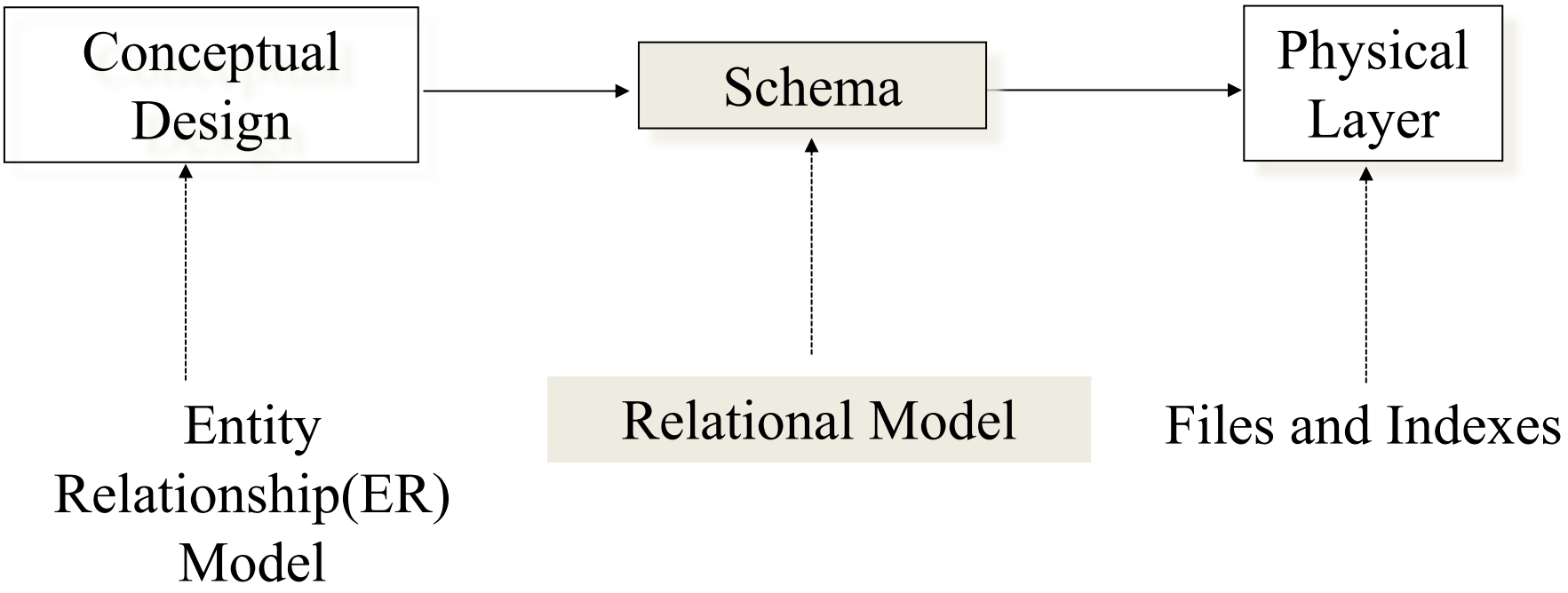
Relational Database Management



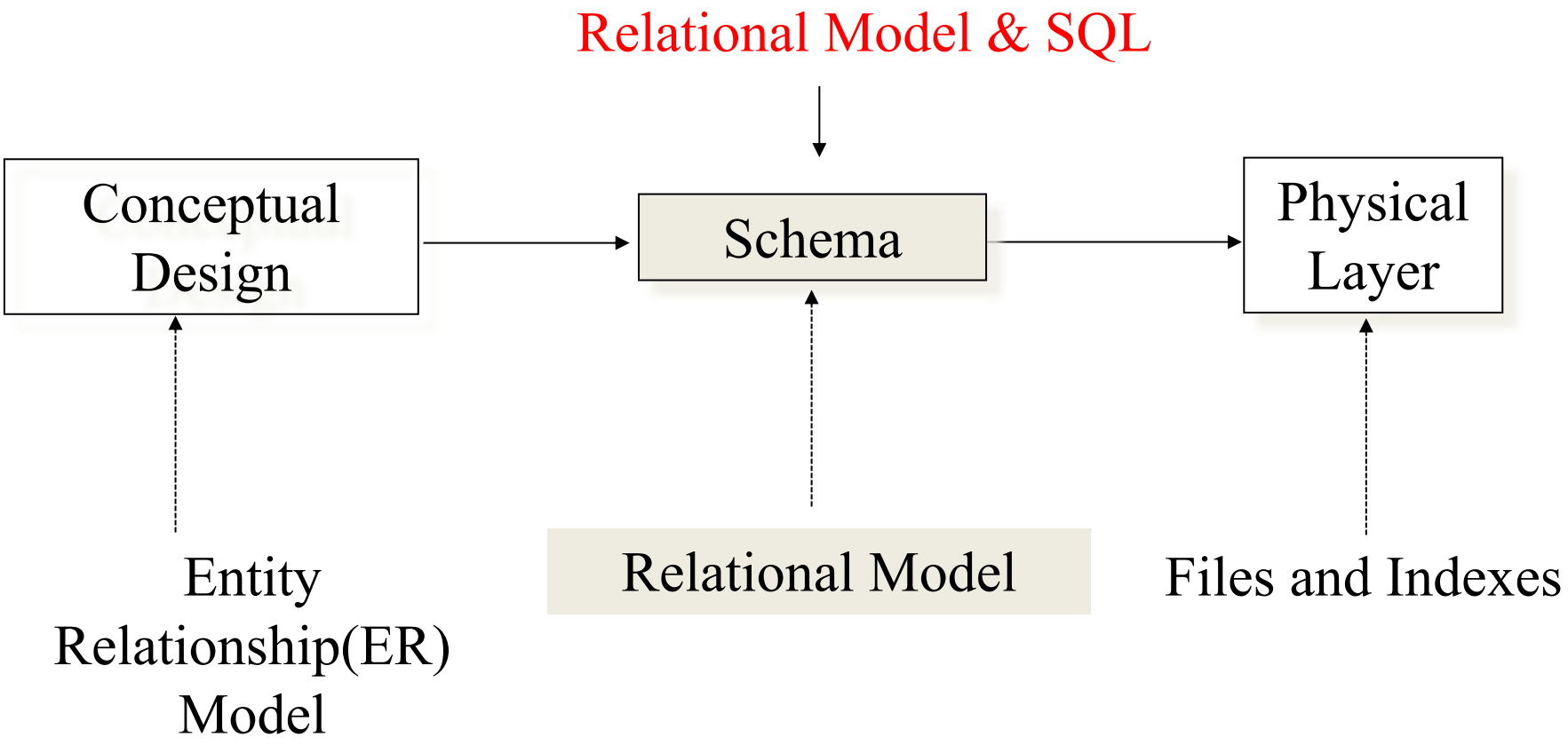
Relational Database Management



Relational Database Management



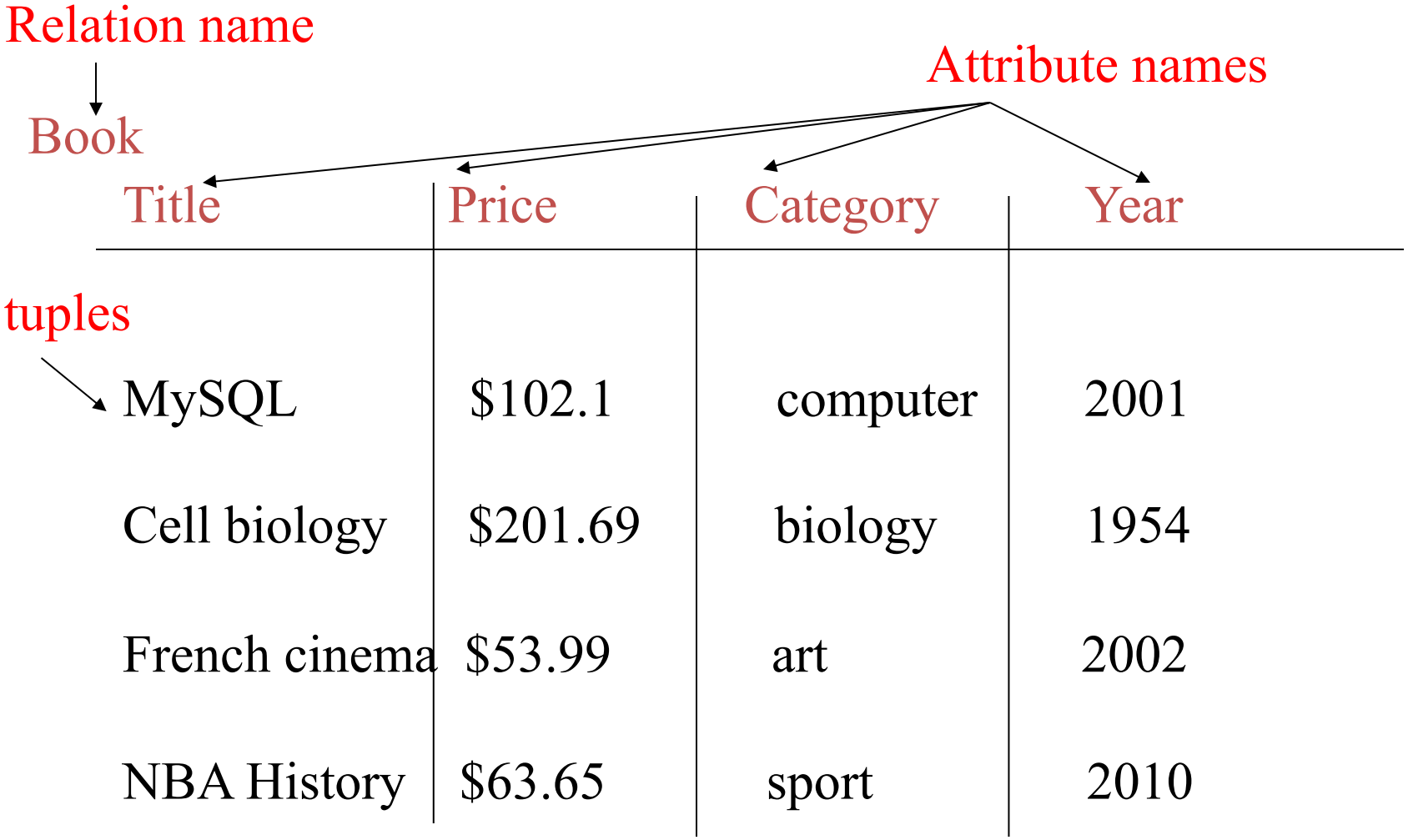
Relational Database Management



Relational Model

- Relational model defines:
  - a way of organizing data: **relations**
  - **operations** to query and/or manipulate the data
- Much easier to use than procedural languages.
  - Say ***what you want*** instead of ***how to do***
- Everything is a relation.
  - Both data and query

Relation: example



Relation

- Attributes
  - Atomic values
  - atomic types: string, integer, real, date, …
- Each relation must have keys
  - Attributes without duplicate values
  - A relation does not contain duplicate tuples.
- *Reordering tuples* does not change the relation.
- *Reordering attributes* does not change the relation.

OSU Oregon State University

Database Schema vs. Database Instance

- Schema of a Relation
  - Names of the relation and their attributes.
  - E.g.: Person (Name, Address, SSN)
  - Types of the attributes
  - Constraints on the values of the attributes
- Schema of the database
  - Set of relation schemata
  - E.g.: Person (Name, Address, SSN)
    Employment(Company, SSN)

Oregon State University

Database Schema vs. Database Instance

- Schema: Book(Title, Price, Category, Year)
- Instance:

Title	Price	Category	Year
MySQL	$102.1	computer	2001
Cell biology	$201.69	biology	1954
French cinema	$53.99	art	2002
NBA History	$63.65	sport	2010

OSU Oregon State University

Formal Relational Query Languages

- Formal languages that express queries over relational schemas.

- Relational Algebra
- Datalog
- Relational calculus

- SQL is compiled to one of these languages.
- Easier to use than SQL to write complex queries and in some application domains.

Relational algebra: operations on relations

- Basic operations:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Cross-product* ($\times$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, … : Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be *composed*. (Algebra is "closed".)

Example Schema

Coffee(cbrand, producer)

CoffeeShop(sname, addr, license)

CoffeeDrinker(dname, addr, phone)

Likes(dname, cbrand)

Sells(sname, cbrand, price)

Frequents(dname, sname)

Oregon State University

Projection

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.

$$\pi_{cbrand}\ Coffee$$

cbrand	producer
Caribou	Baristas
Costa	Coava
Kenya	Delta Coffee

cbrand
Caribou
Costa
Kenya

Selection

- Selects rows that satisfy *selection condition*.
- *Schema* of result identical to schema of the input relation.

sname	cbrand	price
Culture	Costa	2
Interzone	Kenya	4

$$\sigma_{price<3}\ Sells$$

sname	cbrand	price
Culture	Costa	2

- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

OSU
Oregon State
UNIVERSITY

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1.

sname	addr	license
Culture	King Blvd.	201
Interzone	Monroe Ave.	302

dname	sname
John	Culture
Alice	Interzone

$$CoffeeShop \times Frequents$$

sname	addr	license	dname	sname
Culture	Kings Blvd.	201	John	Culture
Interzone	Monroe Ave.	302	John	Culture
Culture	Kings Blvd.	201	Alice	Interzone
Interzone	Monroe Ave.	302	Alice	Interzone

Join

$$R \bowtie_c S = \sigma_c (R \times S)$$

sname	addr	license
Culture	King Blvd.	201
Interzone	Monroe Ave.	302

dname	sname
John	Culture
Alice	Interzone

$$CoffeeShop \bowtie_{CoffeeShop.sname=Frequents.sname} Frequents$$

sname	addr	license	dname	sname
Culture	Kings Blvd.	201	John	Culture
Interzone	Monroe Ave.	302	Alice	Interzone

Join

- *Result schema* same as that of cross-product.
- More meaningful and fewer tuples than cross-product.
- If the condition is equality, the join is called *equi-join*.
- *Natural Join*: Equijoin on *all* common fields.

Datalog

- First created to support recursive queries over relational databases.
- Used in research and industry
  - (probabilistic) learning & inference, data integration, logic programming, distributed processing, …
- We talk about the recursion-free datalog.

Datalog

Movie(mid, title, year, total-gross)
Actor(aid, name, b-year)
Plays(mid, aid)

- Each tuple in database is a *fact*

```
Movie(236878, 'Godfather I', 1972, 40000000)
Movie(879900, 'Godfather II', 1974, 3900000)
Actor(090988,'Robert De Niro', 1943)
```

- Each query is a *rule*

Movies that were produced in 1998 and made more than $2,000.

```
Q1(y):- Movie(x,y,1998,z),z > 2000.
```

Oregon State University

Datalog Example

Movie(mid, title, year, total-gross)
Actor(aid, name, b-year)
Plays(mid, aid)

- Actors who played in a movie whose total gross is more than $2,000.

```
Q2(y):- Actor(x,y,z),Plays(t,x),Movie(t,v,w,f),
        f > 2000.
```

- Actors who played in a movie whose total gross is more than $2,000 and a movie made in 1998.

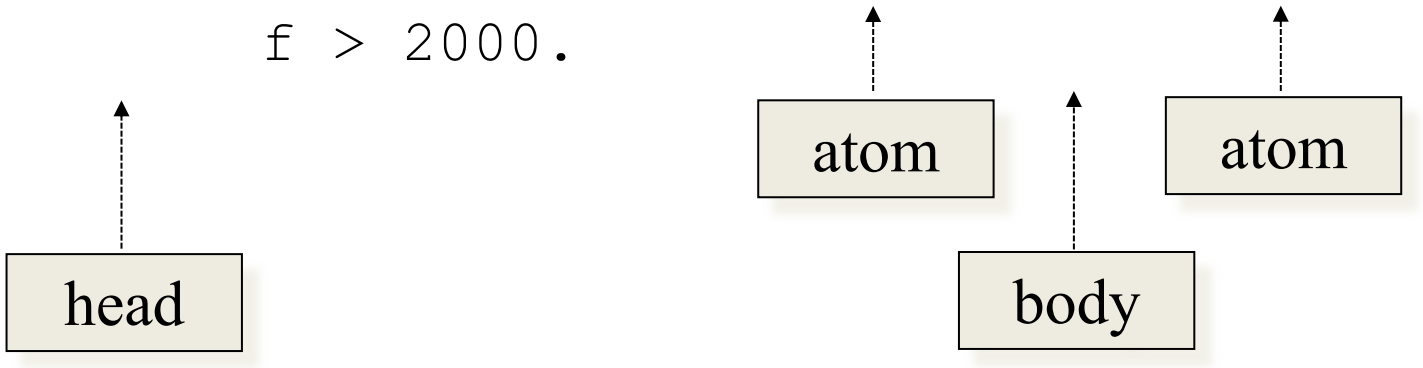
```
Q3(y):- Actor(x,y,z),Plays(t,x),Movie(t,v,w,f),
        f > 2000, Plays(g,x), Movie(g,l,1998,h).
```

Oregon State University

Datalog

Movie(mid, title, year, total-gross)
Actor(aid, name, b-year)
Plays(mid, aid)

```
Q2(y):- Actor(x,y,z),Plays(t,x),Movie(t,v,w,f),
        f > 2000.
```



y: head variable; x, z, t : existential variables

- Extensional Database Predicates (EDB)
  - Movie, Actor, Plays
- Intentional Database Predicate (IDB)
  - Q2

Datalog programs

Movie(mid, title, year, total-gross)
Actor(aid, name, b-year)
Plays(mid, aid)

- A collection of rules: union
- Actors who played in a movie with gross of more than $2,000 or a movie made after 1990.

```
Q4(y):- Actor(x,y,z),Plays(t,x),Movie(t,v,w,f),
      f > 2000.
Q4(y):- Actor(x,y,z),Plays(t,x),Movie(t,v,w,f),
      w > 1990.
```

Datalog with negation

Movie(mid, title, year, total-gross)
Actor(aid, name, b-year)
Plays(mid, aid)

- All actors who did not play in a movie with 'Robert De Niro'.

```
U(x,y,z):- Actor(x,y,z),Plays(t,x),Plays(t,f),
   Actor(f,'Robert De Niro',g).
Q6(y):- Actor(x,y,z), not U(x,y,z).
```

Safe datalog rules

Movie(mid, title, year, total-gross)
Actor(aid, name, b-year)
Plays(mid, aid)

- *Unsafe* rules:

```
V(x,y,z):- Actor(x,y,1998), z > 200.

W(x,y,z):- Actor(x,y,z), not Plays(t,x).
```

- A datalog rule is safe if every variable appear in at least one positive predicate

Relational calculus

- Each relational predicate P is:
  - Atom
  - P $\wedge$ P
  - P $\vee$ P
  - P $\Rightarrow$ P
  - not (P)
  - $\forall x.P$
  - $\exists x.P$
- Each query Q(x1, …, xn) is a predicate P.

Relational calculus

Movie(<u>mid</u>, title, year, total-gross)
Actor(<u>aid</u>, name, b-year)
Plays(<u>mid</u>, <u>aid</u>)

- Actors who played in a movie with total gross of $2,000.

$$Q(x,y,z) = \exists t.\exists f.\exists g.(Actor(x,y,z) \wedge Plays(t,x) \wedge Movie(t,f,g,2000))$$

- Actors who played only in movies produced in 1990.

$$Q(x) = \forall y.Play(y,x) \Rightarrow \exists z.\exists t.Movie(y,z,1990,t)$$

Relational calculus

Movie(<u>mid</u>, title, year, total-gross)
Actor(<u>aid</u>, name, b-year)
Plays(<u>mid</u>, <u>aid</u>)

- Actors who played in some movies with only one actor.

$$Q(x) = \exists y.Play(y,x) \wedge \forall z.\forall t(Play(y,z) \wedge Play(y,t) \Rightarrow z = t)$$

Domain independent RC

Movie(mid, title, year, total-gross)
Actor(aid, name, b-year)
Plays(mid, aid)

- Similar to datalog, one may write *unsafe* queries in RC, called domain dependent.

$$Q(x,y) = not(Actor(x,y,1990))$$

$$Q(x,y) = Plays(x,23412) \vee Plays(y,46372)$$

$$Q(x) = \forall y.Plays(x,y)$$

- RC queries should be domain independent.

Equivalency Theorem

- RA, non-recursive datalog, and RC express the same set of queries.
- They have the same expressive power.

SQL

- A declarative language for querying data stored in relational databases
  - implements RA/RC/Datalog without recursion with slight modifications.
- Many standards: SQL92, SQL99, …
  - We focus on the core functionalities.

The Basic Form

SELECT returned attribute(s)

FROM relation(s)

WHERE conditions on the tuples of the table(s)

One or more

1. Apply the WHERE clause's conditions on all relations in the tables in the FROM clause.
2. Return the values of the attributes in the SELECT clause.

OSU Oregon State University

Example Schema

Coffee(cbrand, producer)

CoffeeShop(sname, addr, license)

CoffeeDrinker(dname, addr, phone)

Likes(dname, cbrand)

Sells(sname, cbrand, price)

Frequents(dname, sname)

Single Relation Query

What brands are made by Baristas?

```
SELECT cbrand
FROM Coffee
WHERE producer = 'Baristas';
```

cbrand	producer
Caribou	Baristas
Costa	Coava
Kenya	Delta Coffee

cbrand
Caribou

Oregon State University

Using *

What coffee brands are produced by Baristas?

```
SELECT *
 FROM Coffee
 WHERE producer = 'Baristas';
```

cbrand	producer
Caribou	Baristas
Costa	Coava
Kenya	Delta Coffee

cbrand	producer
Caribou	Baristas

Oregon State University

WHERE clause

- May have complex conditions
- Logical operators: OR, AND, NOT
- Comparison operators: <, >, =, <>,…
- Types specific operators: LIKE, …

Null Values

- Some tuples may not contain any value for some of their attributes
  - The operator did not enter the data
  - The operator did not know the value
  - …
- Ex: We do not know Fred's salary.
  - Put 0.0 ➔ Fred is not on unpaid leave!
- Databases use null value for these cases

OSU Oregon State University

A value not like any other value!

- A tuple in Sells relation:

sname	cname	price
Culture	Kenya	NULL

```
SELECT *
FROM Sells
WHERE price < 0.0 OR price >= 0.0
```

Does not return Culture.

A value not like any other value!

- A tuple in Sells relation:

sname	cname	price
Culture	Kenya	NULL

```
SELECT *
FROM Sells
WHERE price IS NULL
```

Multi Relation Query: Join

- Find relations between different types of entities.

- Using relations *Likes(dname, cname)* and *Frequents(dname, sname)*, find the coffee brands liked by at least one person who frequents *Culture*.

```
SELECT cname
 FROM   Likes, Frequents
 WHERE Frequents.sname = 'Culture'
   AND Frequents.dname = Likes.dname;
```

Oregon State University

Outer join

- By default, a join returns only tuples that satisfy the join condition.
- **Left outer join** returns all tuples from the left relation.
  - The left outer join of *CofffeeShop* and *Frequents* on *sname*

sname	addr	license
Culture	King Blvd.	201
Interzone	Monroe Ave.	302

dname	sname
John	Culture

sname	addr	license	dname	sname
Culture	Kings Blvd.	201	John	Culture
Interzone	Monroe Ave.	302	null	null

  - Each RDBMS may have a different syntax, MySQL:

```
SELECT * FROM CoffeeShop
 LEFT JOIN Frequents
 ON CoffeeShop.sname = Frequents.sname;
```

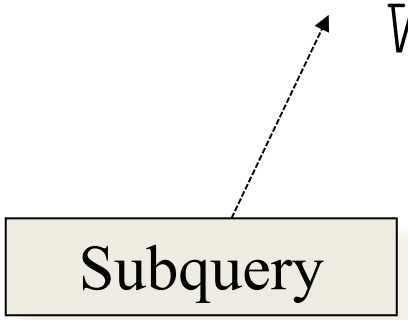
- Right (full) outer join retains all tuples from the right (all) relation.

Subqueries

- SQL queries that appear in WHERE or FROM parts of another query.
- Using *Sells(sname, cname, price)*, find the coffee shops that serve *Costa* for the same price *Culture* charges for *Kenya*.
  - Figure out Culture's price for Kenya : *Kenya-Price*
  - Find shops that offer Costa at price = *Kenya-Price*

Subqueries

```
SELECT sname
FROM Sells
WHERE cname='Costa' AND price=
      (SELECT price
       FROM Sells
       WHERE  sname = 'Culture'
          AND cname = 'Kenya');
```



Subqueries: ALL, ANY

- One may want to compare a value to a set of values

- Using *Sells(sname, cname, price)*, find the coffee shops that serve *Kenya* for a cheaper price than the price that all coffee shops charge for *Costa*.
  - Figure out the set of all prices for *Costa* : *CostaPrice.*
  - Find the bars that offer *Kenya* at a cheaper price than all values in *CostaPrice.*

Subqueries: ALL, ANY

```
SELECT sname
 FROM Sells
 WHERE cname='Kenya' AND
  price < ALL
                (SELECT price
                 FROM Sells
                 WHERE cname='Costa');
```

- ANY instead of ALL:
  - Returns the coffee shops that serve *Kenya* for a cheaper price than the price that **at least one coffee shop** charges for *Costa*.

Subqueries: IN

- One may like to check if the result of a subquery contains a particular value.
- Using *Coffee(cname, producer)* and *Likes(dname, cname)* find the producers of the coffee brands *John* likes.

```
SELECT producer
FROM Coffee
WHERE cname IN
            (SELECT cname
             FROM Likes
             WHERE dname='John');
```

A set of coffee brands

Subqueries: Not IN

- Negation of IN
- One may use it to check if a set does not contain a certain value.
- Used similar to IN

Subqueries: Exists, Not Exists

- We may want to check if a subquery has any result.
- Using *Coffee(cname, producer)*, find the coffee brands that are the **only** brand made by their producers.

```
SELECT cname
 FROM Coffee c1
 WHERE NOT EXISTS
       (SELECT *
        FROM Coffee
        WHERE producer = b1.producer
          AND cname <> c1.cname);
```

Bag versus Set

- Duplicates are allowed in bags.
  - {a, a, b, b, b} vs. {a, b}
- Generally, the results of SQL queries are bags.

```
SELECT producer
FROM Coffee;
```

cbrand	producer
Caribou	Baristas
Costa	Coava
Kenya	Baristas

producer
Baristas
Coava
Baristas

Removing Duplicates

- Use DISTINCT

```
SELECT DISTINCT producer
 FROM Coffee;
```

cbrand	producer
Caribou	Baristas
Costa	Coava
Kenya	Baristas

producer
Baristas
Coava

Set Operations

- R UNION S
  - Returns the union between tuples of relation R and tuples of relation S.
- R INTERSECT S
  - Returns the tuples common between relation R and relation S.
- R EXCEPT S
  - Returns the tuples found in relation R but not in relation S.
  - Can be expressed using NOT IN.

Set Operations: Example

- Using relations *Likes(dname, cname), Sells(sname, cname, price),* and *Frequents(dname, sname),* find the coffee drinkers and brands such that
  - The coffee drinker likes the brand, **and**
  - The coffee drinker frequents at least one coffee shop that sells the brand

- "**and**" indicates that we should compute intersection.

Set operations: Example

```
(SELECT * FROM Likes)
  INTERSECT
(SELECT dname, cname
  FROM Frequents, Sells
 WHERE Frequents.sname = Sells.sname);
```

The drinker likes the brand

The drinker frequents at the coffee shop that serves the brand

Set Operations

- The results of set operations in SQL do not have any duplicate tuples.
- We can force them not to remove duplicates by ALL.
  - .. INTERSECT … ➔ .. INTERSECT ALL …
  - .. UNION … ➔ .. UNION ALL …
  - .. DIFFERENCE … ➔ .. DIFFERENCE ALL …

Oregon State University

Aggregation functions

- Compute some value based on the values of an attribute.
  - Example functions: **Count**, **Sum**, **Avg**, **Min**, **Max**
  - Each RDBMS may define additional functions.

- Using *Coffee(cname, producer)*, find the number of coffee brands in the database.

```
Select Count(cname)
From Coffee;
```

Aggregation functions

- Using *Distinct,* aggregation functions ignore duplicates.
- Using *Sell(sname, cname, price)*, find the number of coffee shops that serve Costa.

```
Select Count( Distinct sname)
From Sell
Where cname =‘Costa’;
```

Oregon State University

Aggregation functions

Generally, aggregation functions do **not** consider NULL values.

The number of **priced brands** sold by Culture.

```
Select Count(price)
From Sells
Where sname='Culture';
```

The number of **coffee brands** sold by Culture.

```
Select Count(cname)
From Sells
Where sname='Culture';
```

The number of **coffee brands** sold by Culture.

```
Select Count(*)
From Sells
Where sname='Culture';
```

OSU Oregon State University

Aggregation functions over groups

- One often likes to compute aggregation functions for groups of tuples.
- Using *Sells(sname, cname, price)* find the minimum price of each coffee brand.
  - Group tuples in *Sells* based on coffee brand.
  - Compute *Min* over the prices in each group of tuples.

sname	cname	price
Culture	Kenya	9
Interzone	Costa	10
Culture	Fancy	15
Interzone	Kenya	10
Ava	Fancy	20
Ava	Kenya	16
Culture	Costa	5
Tried & True	Kenya	10

cname	minprice
Kenya	9
Fancy	15
Costa	5

Group by

- Using *Sells(sname, cname, price)* find the minimum price of each coffee brand.

```
Select cname, Min(price) As minprice
From Sells
Group By cname;
```

optional

sname	cname	price
Culture	Kenya	9
Interzone	Costa	10
Culture	Fancy	15
Interzone	Kenya	10
Ava	Fancy	20
Ava	Kenya	16
Culture	Costa	5
Tried & True	Kenya	10

cname	minprice
Kenya	9
Fancy	15
Costa	5

Group by

- We may use multiple attributes for grouping.
- The attributes in the *Select* clause are either *aggregated values* or *attributes in the Group By clause*.

```
Select cname, Min(price),sname
 From Sells
 Group By cname;
```

  - Exceptions in some RDBMS, e.g., MySQL 5.7.

Generally, *Group By* does **not** sort the groups.

  - There are exceptions, e.g., older versions of MySQL, but do not trust them!

Oregon State University

Grouping attributes from different relations.

- Using *Likes(dname, cname)* and *Sells(sname, cname, price),* for each coffee drinker find the minimum price of every brand he/she likes.

```
Select dname, cname, Min(price) As minprice
From Likes, Sells
Where Likes.cname = Sells.cname
Group By dname, cname;
```

sname	cname	price
Culture	Kenya	9
Interzone	Costa	10
Culture	Fancy	15
Interzone	Kenya	10
Ava	Fancy	20
Ava	Kenya	16
Culture	Costa	5
Tried & True	Kenya	10

dname	cname
John	Kenya
Ben	Costa
Smith	Fancy
John	Costa

dname	cname	minprice
John	Kenya	9
John	Costa	5
Ben	Costa	5
Smith	Fancy	15

Oregon State University

Filtering groups

- We may filter out some groups using their attributes' values.

```
Select cname, Min(price) As minprice
  From Sells
  Where sname = 'Interzone' OR sname = 'Ava'
 Group By cname;
```

sname	cname	price
Culture	Kenya	9
Interzone	Costa	10
Culture	Fancy	15
Interzone	Kenya	10
Ava	Fancy	20
Ava	Kenya	16
Culture	Costa	5
Tried & True	Kenya	10

cname	minprice
Kenya	10
Fancy	20
Costa	10

Filtering groups based on aggregated values

- Using *Sells(sname, cname, price)*, find the **minimum price** of each brand whose **maximum price** is less than 11.

sname	cname	price
Culture	Kenya	9
Interzone	Costa	10
Culture	Fancy	15
Interzone	Kenya	10
Ava	Fancy	20
Ava	Kenya	16
Culture	Costa	5
Tried & True	Kenya	10

cname	minprice
Costa	5

```
Select cname, Min(price) As minprice
From Sells
Where Max(price) < 11
Group By cname
```

Oregon State University

Having clause

- We use *Having* clauses to filter out groups based on their aggregated values.

```
Select cname, Min(price) As minprice
From Sells
Group By cname
Having Max(price) < 11
```

sname	cname	price
Culture	Kenya	9
Interzone	Costa	10
Culture	Fancy	15
Interzone	Kenya	10
Ava	Fancy	20
Ava	Kenya	16
Culture	Costa	5
Tried & True	Kenya	10

cname	minprice
Costa	5

Oregon State University

Having clause

- We may use aggregated values over attributes other than the ones in the *Group By* clause.
- Using *Sells(sname, cname, price)*, find the **minimum price** of each brand served in more than three coffee shops.

sname	cname	price
Culture	Kenya	9
Interzone	Costa	10
Culture	Fancy	15
Interzone	Kenya	10
Ava	Fancy	20
Ava	Kenya	16
Culture	Costa	5
Tried & True	Kenya	10

cname	minprice
Kenya	5

```
Select cname, Min(price) As minprice
From Sells
Group By cname
Having Count(sname) > 3
```

Having clause may act as a *Where* clause

- Using *Sells(sname, cname, price)*, find the minimum price of *Kenya* or brands whose maximum price is less than 11.

```
Select cname, Min(price)
 From Sells
 Group By cname
 Having (Max(price) < 11) Or (cname='Kenya')
```

- It works only for the attributes in the *Group By* clause.

```
Select cname, Min(price)
 From Sells
 Group By cname
 Having (Max(price) < 11) Or (sname='Red Lion')
```

Oregon State University

Sorting the output

- Using *Sells(sname, cname, price)*, find the **minimum price** of each brand whose **maximum price** is less than 11 and sort the results according to brands' names.

```
Select cname, Min(price) As minprice
 From Sells
 Group By cname
 Having Max(price) >= 11
 Order By cname;
```

- One may use *Desc* to change the sort order.

```
Order By cname Desc;
```

- You may use *Order By* without *Group By* and *Having*.