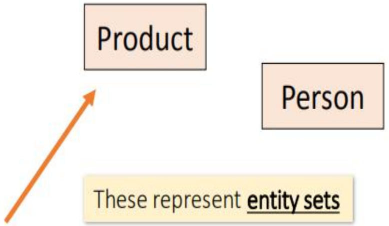
Ders02 - ER Model:

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*

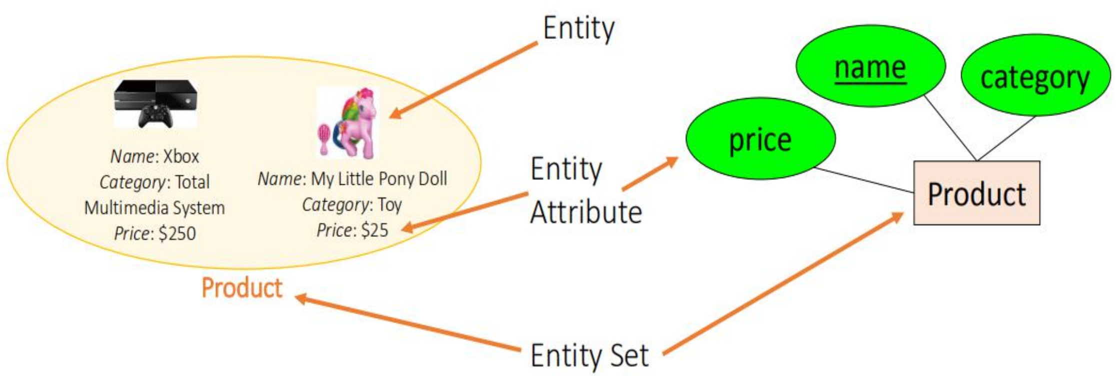


These represent **entity sets**

Entities vs. Entity Sets

Example:

Entities are not explicitly represented in E/R diagrams!



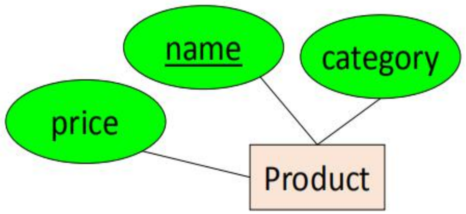
Keys

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

Denote elements of the primary key by underlining.

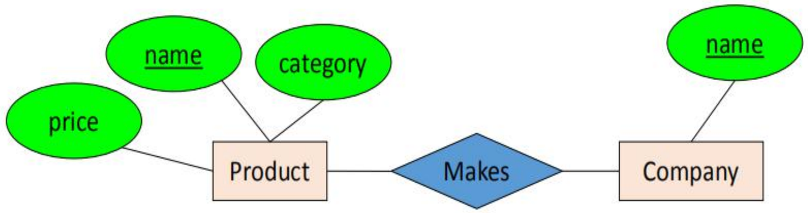
Here, {price, category} is not a key.

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

What is a Relationship?



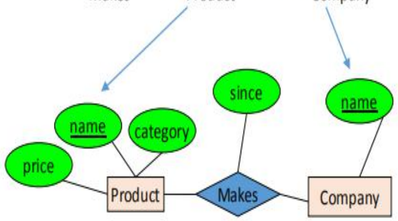
A relationship between **entity sets P and C** is a ***subset of all possible pairs of entities in P and C***, with tuples uniquely identified by ***P and C's keys***

What is a Relationship?

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

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

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



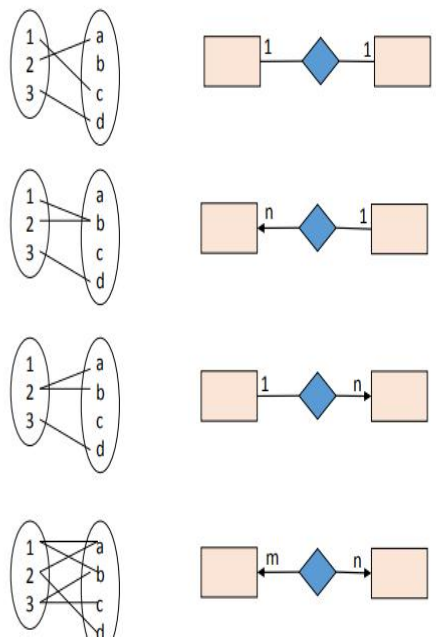
Multiplicity of E/R Relationships

One-to-one:

Many-to-one:

One-to-many:

Many-to-many:

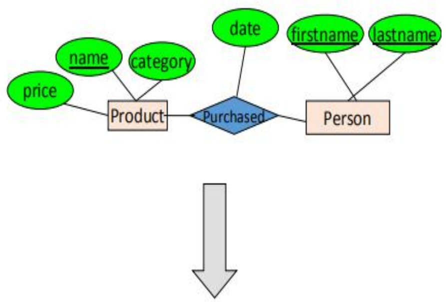


Indicated using arrows

X -> Y means __there exists a function mapping from X to Y__ *(recall the definition of a function)*

From E/R Diagrams to Relational Schema

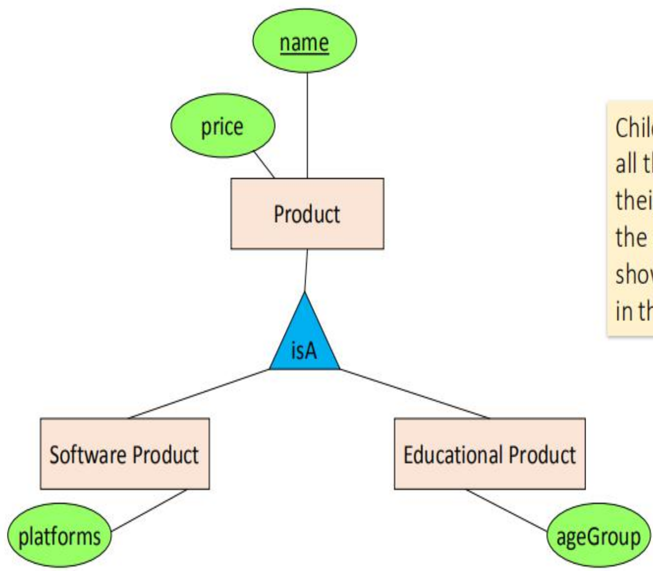
```
CREATE TABLE Purchased(
  name      CHAR(50),
  firstname CHAR(50),
  lastname  CHAR(50),
  date      DATE,
  PRIMARY KEY (name, firstname, lastname),
  FOREIGN KEY (name)
        REFERENCES Product,
  FOREIGN KEY (firstname, lastname)
        REFERENCES Person
)
```



Purchased

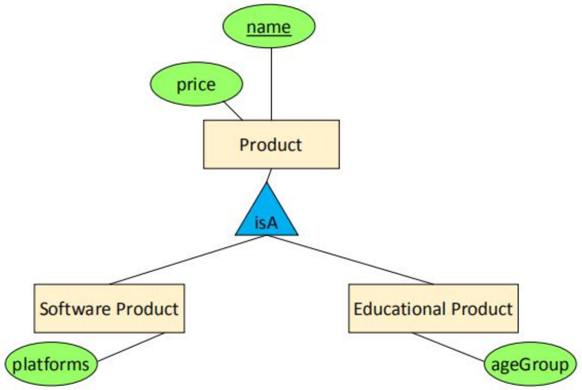
name	firstname	lastname	date
Gizmo1	Bob	Joe	01/01/15
Gizmo2	Joe	Bob	01/03/15
Gizmo1	JoeBob	Smith	01/05/15

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

Think like tables…



Product

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

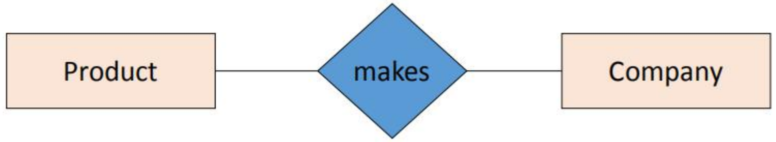
Sw.Product

name	platforms
Gizmo	unix

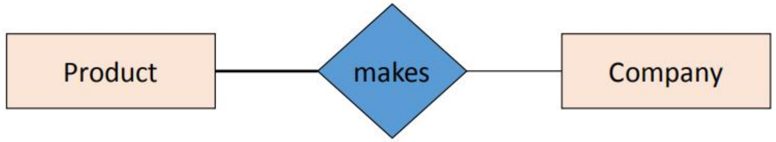
Ed.Product

name	ageGroup
Gizmo	todler
Toy	retired

Participation Constraints: Partial v. Total



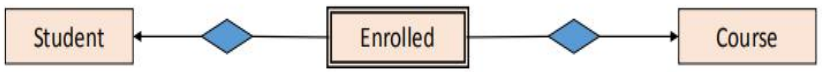
Are there products made by no company?
Companies that don't make a product?



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

Weak Entity Sets

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



Equal to:

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

Der03 - SQL:

❖ 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!

❖ 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(Pname: string, Price: float, Category: string, Manufacturer: string)

Tables in SQL

Product

Attribute || Column

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

Tuple || Row

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

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

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!

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

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.

❖ A Few Details

- ✓ SQL statements are case insensitive: “Same: SELECT, Select, select” or “Same: Product, product”
- ✓ Values are not: “Different: ‘Seattle’, ‘seattle’ ”
- ✓ Use single quotes for constants: ‘abc’ - yes but “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.

Foreign Key constraints

- Suppose we have the following schema:

> Students(<u>sid</u>: *string*, name: *string*, gpa: *float*)
>
> Enrolled(<u>student_id</u>: *string*, <u>cid</u>: *string*, grade: *string*)

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

Students

sid	name	gpa
101	Bob	3.2
123	Mary	3.8

Enrolled

student_id	cid	grade
123	564	A
123	537	A+

student_id alone is not a key- what is?

We say that student_id is a <u>**foreign key**</u> that refers to Students

Declaring Foreign Keys

> Students(<u>sid</u>: *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)
)
```

Joins

> Product(<u>PName</u>, Price, Category, Manufacturer)
>
> Company(<u>CName</u>, 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
```

Tuple Variable Ambiguity in Multi-Table

> Person(<u>name</u>, address, worksfor)
>
> Company(<u>name</u>, 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
```

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?

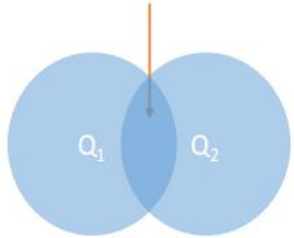
Explicit Set Operators: INTERSECT

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

INTERSECT

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

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



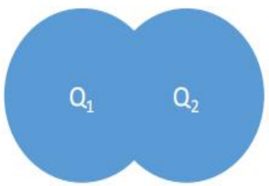
UNION

```
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?

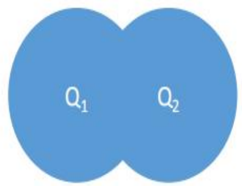
UNION ALL

```
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
```

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

What is the multiset version?

INTERSECT: Remember the semantics!

Company(<u>name</u>, hq_city) AS C
Product(<u>pname</u>, 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(<u>name</u>, hq_city)
Product(<u>pname</u>, 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?

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

ANY and ALL not supported by SQLite.

You can also use operations of the form:

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

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)
```

Correlated Queries

Movie(<u>title, year,</u> 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)

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
```

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.

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…***

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)
```

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

Equivalent

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

A <u>**universal quantifier**</u> is of the form "for all"

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

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!

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!

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

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

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

Ders04 - Relational Algebra:

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*

RDBMS Architecture

How does an SQL engine work ?

SQL Query → Relational Algebra (RA) Plan → *Optimized* RA Plan → Execution

- SQL Query: Declarative query (from user)
- Relational Algebra (RA) Plan: Translate to relational algebra expresson
- *Optimized* RA Plan: *Find logically equivalent- but more efficient- RA expression*
- Execution: Execute each operator of the optimized plan!

Relational Algebra (RA)

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

- Derived or auxiliary operators:
  - 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…

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:

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

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:

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

Cross-Product ($\times$)

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

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

SQL:

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

RA:

$$Students \times People$$

Renaming ($\rho - Rho$)

- 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*

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:

$$Students \bowtie People$$

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?

Division

- Notation: $r \div s$
- 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 $r \div s$ is a relation on schema

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

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

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

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

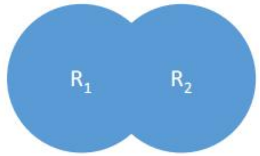
s

- $r \div s$

A	B	C
α	a	γ
γ	a	γ

Union (∪) and Difference (–)

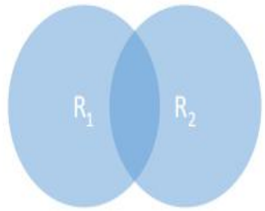
- R1 ∪ R2
- Example:
  - ActiveEmployees ∪ RetiredEmployees



- 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 ($\bowtie_\theta$)

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

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

SQL:

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

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

RA:

$$Students \bowtie_\theta People$$

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

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

- 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

SQL:

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

Most common join in practice!

RA:

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

Semijoin (⋉)

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

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

SQL:

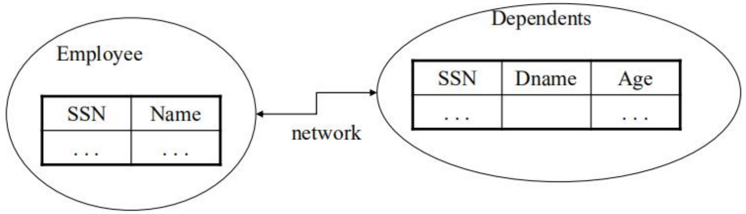
```
SELECT DISTINCT
  sid,sname,gpa
FROM
  Students,People
WHERE
  sname = pname;
```

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!

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

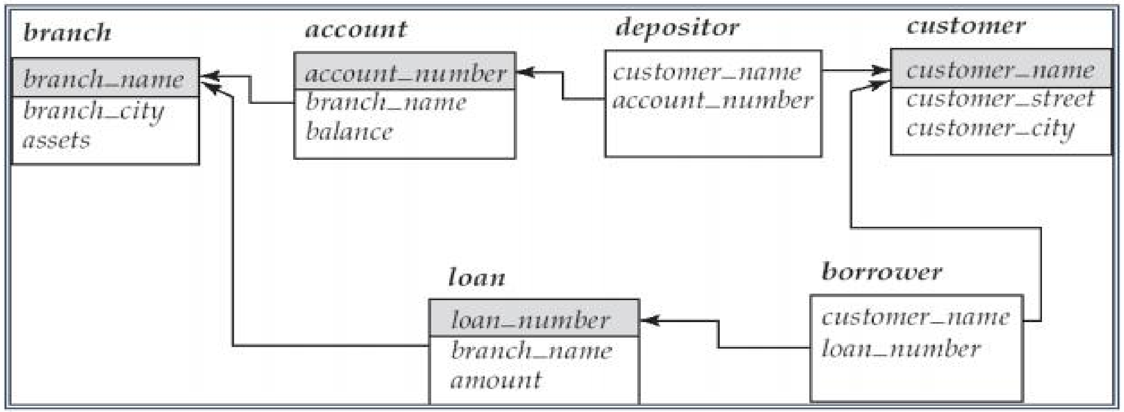
$R = \text{Employee} \ltimes T$

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

EXAMPLE RELATIONAL ALGEBRA

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)



- Find all loans of over $1200

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

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

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

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

$$\Pi_{customer\_name} (\sigma_{branch\_name = \text{“Perryridge”}} (\sigma_{borrower.loan\_number = loan.loan\_number} (borrower \times loan)))$$

- 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.

$$\Pi_{customer\_name} (\sigma_{branch\_name = \text{“Perryridge”}} (\sigma_{borrower.loan\_number = loan.loan\_number} (\text{borrower x loan}))) - \Pi_{customer\_name}(\text{depositor})$$

- 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.

$$\Pi_{customer\_name} (borrower) \cap \Pi_{customer\_name} (depositor)$$

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

$$\Pi_{\text{customer\_name}} (\sigma_{\text{branch\_name = “Perryridge”}} (\sigma_{\text{borrower.loan\_number = loan.loan\_number}} (\text{borrower x loan})))$$

  - **Query 2**

$$\Pi_{\text{customer\_name}}(\sigma_{\text{loan.loan\_number = borrower.loan\_number}} ((\sigma_{\text{branch\_name = “Perryridge”}} (\text{loan})) \text{ x borrower}))$$

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

$$\Pi_{customer\_name,\ branch\_name} (depositor \bowtie account) \div \Pi_{branch\_name} (\sigma_{branch\_city = \text{“Brooklyn”}} (branch))$$