Principles of Database Systems
CSE 544

Lecture #2

SQL,
Relational Algebra,
Relational Calculus

Announcements

- Makeup:
  - Friday, March 30, 11-12:30, Room TBD
  - 1st Paper review due (Answering Queries Using Views, Sec.1-3)
- Regular lecture:
  - Monday, April 2nd, before class
  - 2nd Paper review due (What Goes UP, skip sections 5-7)
- Cancelled:
  - Lecture on Wednesday, April 4

- Subscribe to the mailing list!
  - If you haven't received yesterday's email, then you aren't subscribed yet
- Still waiting to register for the class?
  - Send me an email and I will register you

Outline

- Finish SQL: NULLs, Grouping/aggregation
- Relational Calculus
- Relational Algebra

They are equivalent and why we care

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

Null Values

height unknown

Person(<u>name</u>, age, height, weight)

```
INSERT INTO Person VALUES('Joe',20,NULL,200)
```

Rules for computing with NULLs

- If x is NULL then 4*(3-x)/7 is still NULL
- If x is 2 then x>5 is FALSE
- If x is NULL then x>5 is UNKNOWN
- If x is 10 then x>5 is TRUE

FALSE	=	0
UNKNOWN	=	0.5
TRUE	=	1

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 OR weight > 190)
```

E.g.
age=20
height=NULL
weight=200

Rule in SQL: include only tuples that yield TRUE

Null Values

Unexpected behavior:

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

Some Persons 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 all Person in included

Detour into DB Research

Imielinski&Libski, *Incomplete Databases*, 1986

- Database = is in one of several states, or *possible worlds*
  - Number of possible worlds is exponential in size of db
- Query semantics = return the *certain answers*

Very influential paper:

- Incomplete DBs used in probabilistic databases, *what-if* scenarios, data cleaning, data exchange

In SQL, NULLs are the simplest form of incomplete database:

- Database = a NULL takes independently any possible value
- Query semantics = not exactly certain answers (why?)

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

Outerjoins

An "inner join":

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

Same as:

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

But Products that never sold will be lost !

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

Outerjoins

If we want the never-sold products, need a "left outer join":

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

Product(<u>name</u>, category)
Purchase(prodName, store)

Product

<u>name</u>	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

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 both left and right tuples even if there's no match

Aggregations

Five basic aggregate operations in SQL

- count
- sum
- avg
- max
- min

Purchase(product, price, quantity)

Counting Duplicates

COUNT applies to duplicates, unless otherwise stated:

```
SELECT  count(product)
FROM    Purchase
WHERE   price>3.99
```

Same as count(*)

Except if some product is NULL

We probably want:

```
SELECT  count(DISTINCT product)
FROM    Purchase
WHERE   price>3.99
```

Purchase(product, price, quantity)

Grouping and Aggregation

Find total quantities for all sales over $1, by product.

```
SELECT      product, sum(quantity) AS TotalSales
FROM        Purchase
WHERE       price > 1
GROUP BY    product
```

product	price	quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

What is the answer?

Grouping and Aggregation

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: group attrs and aggregates.

1&2. FROM-WHERE-GROUPBY

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	~~0.5~~	~~50~~
Banana	2	10
Banana	4	10

```
SELECT      product, sum(quantity) AS TotalSales
FROM        Purchase
WHERE       price > 1
GROUP BY    product
```

3. SELECT

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	~~0.5~~	~~50~~
Banana	2	10
Banana	4	10

Product	TotalSales
Bagel	40
Banana	20

```
SELECT     product, sum(quantity) AS TotalSales
FROM       Purchase
WHERE      price > 1
GROUP BY   product
```

Ordering Results

```
SELECT product, sum(quantity) as TotalSales
FROM   purchase
GROUP BY product
ORDER BY TotalSales DESC
LIMIT 20                                    -- postgres onl
```

```
SELECT product, sum(quantity) as TotalSales
FROM   purchase
GROUP BY product
ORDER BY sum(quantity) DESC
LIMIT 20                                    -- postgres only
```

Equivalent, but not all systems accept both syntax forms

HAVING Clause

Same query as earlier, except that we consider only products that had at least 30 sales.

```
SELECT      product, sum(quantity)
FROM        Purchase
WHERE       price > 1
GROUP BY    product
HAVING      count(*) > 30
```

HAVING clause contains conditions on aggregates.

WHERE vs HAVING

- WHERE condition: applied to individual rows
  - Determine which rows contributed to the aggregate
  - All attributes are allowed
  - No aggregates functions allowed
- HAVING condition: applied to the entire group
  - Entire group is returned, or not al all
  - Only group attributes allowed
  - Aggregate functions allowed

General form of Grouping and Aggregation

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

Why ?

S = may contain attributes $a_1,\ldots,a_k$ and/or any aggregates but NO OTHER ATTRIBUTES

C1 = is any condition on the attributes in $R_1,\ldots,R_n$

C2 = is any condition on aggregate expressions and on attributes $a_1,\ldots,a_k$

Semantics of SQL With Group-By

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

Evaluation steps:

1. Evaluate FROM-WHERE using Nested Loop Semantics
2. Group by the attributes $a_1,\ldots,a_k$
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Purchase(product, price, quantity)

Product(pname, manufacturer)

Empty Groups

- In the result of a group by query, there is one row per group in the result
- A group can never be empty!
- In particular, count(*) is never 0

```
SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.pname = y.product
GROUP BY x.manufacturer
```

What if there are no purchases for a manufacturer

Empty Group Problem

Purchase(product, price, quantity)

Product(<u>pname</u>, manufacturer)

```
SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.pname = y.product
GROUP BY x.manufacturer
```

What if there are no purchases for a manufacturer

Empty Group Solution: Outer Join

Purchase(product, price, quantity)

Product(<u>pname</u>, manufacturer)

```
SELECT x.manufacturer, count(y.product)
FROM Product x LEFT OUTER JOIN Purchase y
ON x.pname = y.product
GROUP BY x.manufacturer
```

Relational Query Languages

1. Relational Algebra

2. Recursion-free datalog with negation
   - This is the core of SQL, cleaned up

3. Relational Calculus

These three formalisms express the same class of queries

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Running Example

Find all actors who acted both in 1910 and in 1940:

```
Q: SELECT DISTINCT a.fname, a.lname
   FROM   Actor a, Casts c1, Movie m1, Casts c2, Movie m2
   WHERE  a.id = c1.pid   AND c1.mid = m1.id
     AND  a.id = c2.pid   AND c2.mid = m2.id
     AND  m1.year = 1910 AND m2.year = 1940;
```

Two Perspectives

- Named Perspective:
  Actor(id, fname, lname)
  Casts(pid,mid)
  Movie(id,name,year)
- Unnamed Perspective:
  Actor = arity 3
  Casts = arity 2
  Movie = arity 3

1. Relational Algebra

Used internally by RDBMs to execute queries

The Basic Five operators:

- Union: $\cup$
- Difference: -
- Selection: σ
- Projection: Π
- Join: $\bowtie$

Renaming: ρ (for named perspective)

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

1. Relational Algebra (Details)

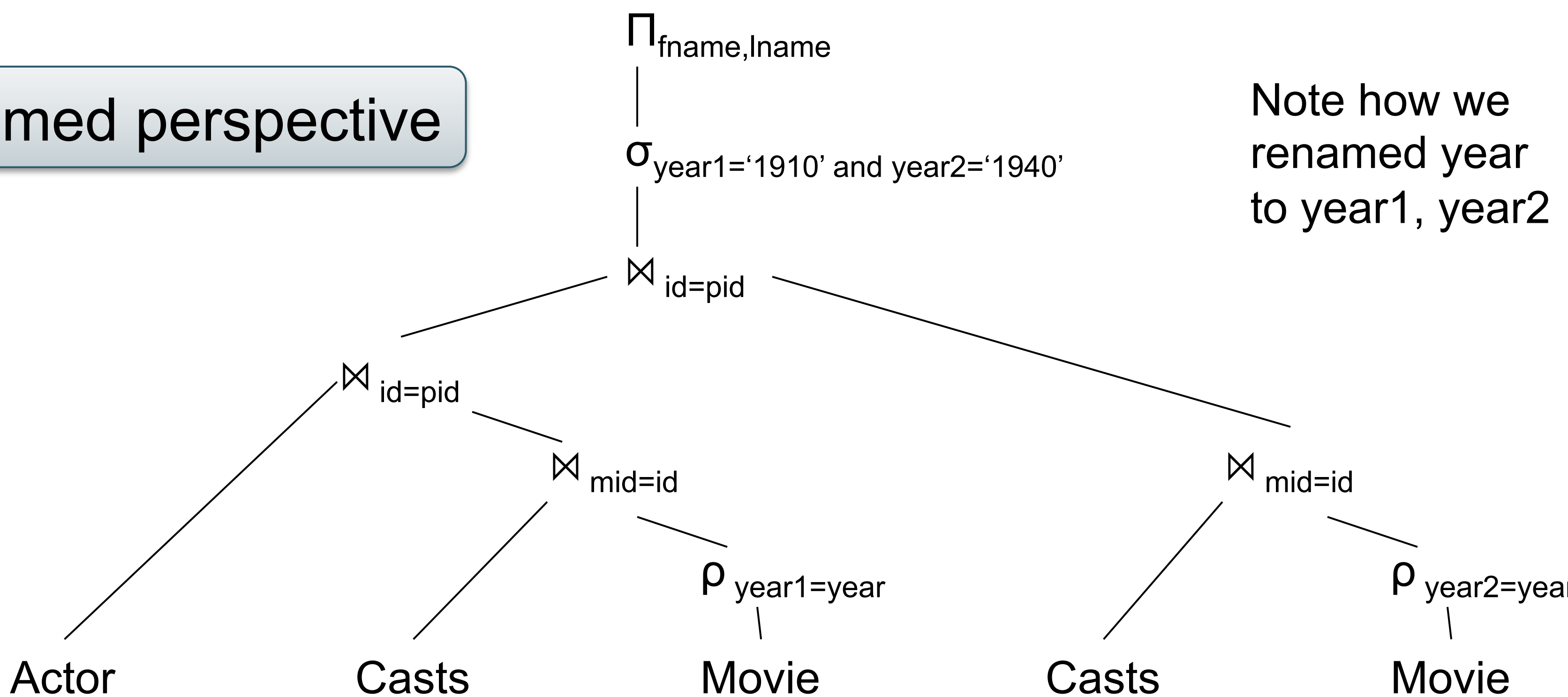
- Selection: returns tuples that satisfy condition
  - Named perspective: $\sigma_{year = '1910'}$(Movie)
  - Unnamed perspective: $\sigma_{3 = '1910'}$ (Movie)
- Projection: returns only some attributes
  - Named perspective: $\Pi_{fname,lname}$(Actor)
  - Unnamed perspective: $\Pi_{2,3}$(Actor)
- Join: joins two tables on a condition
  - Named perspective: Casts $\bowtie_{mid=id}$ Movie
  - Unnamed perspectivie: Casts $\bowtie_{2=1}$ Movie

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

1. Relational Algebra

```
Q: SELECT DISTINCT a.fname, a.lname
   FROM   Actor a, Casts c1, Movie m1, Casts c2, Movie m2
   WHERE  a.id = c1.pid      AND c1.mid = m1.id
     AND  a.id = c2.pid      AND c2.mid = m2.id
     AND  m1.year = 1910     AND m2.year = 1940;
```

Named perspective



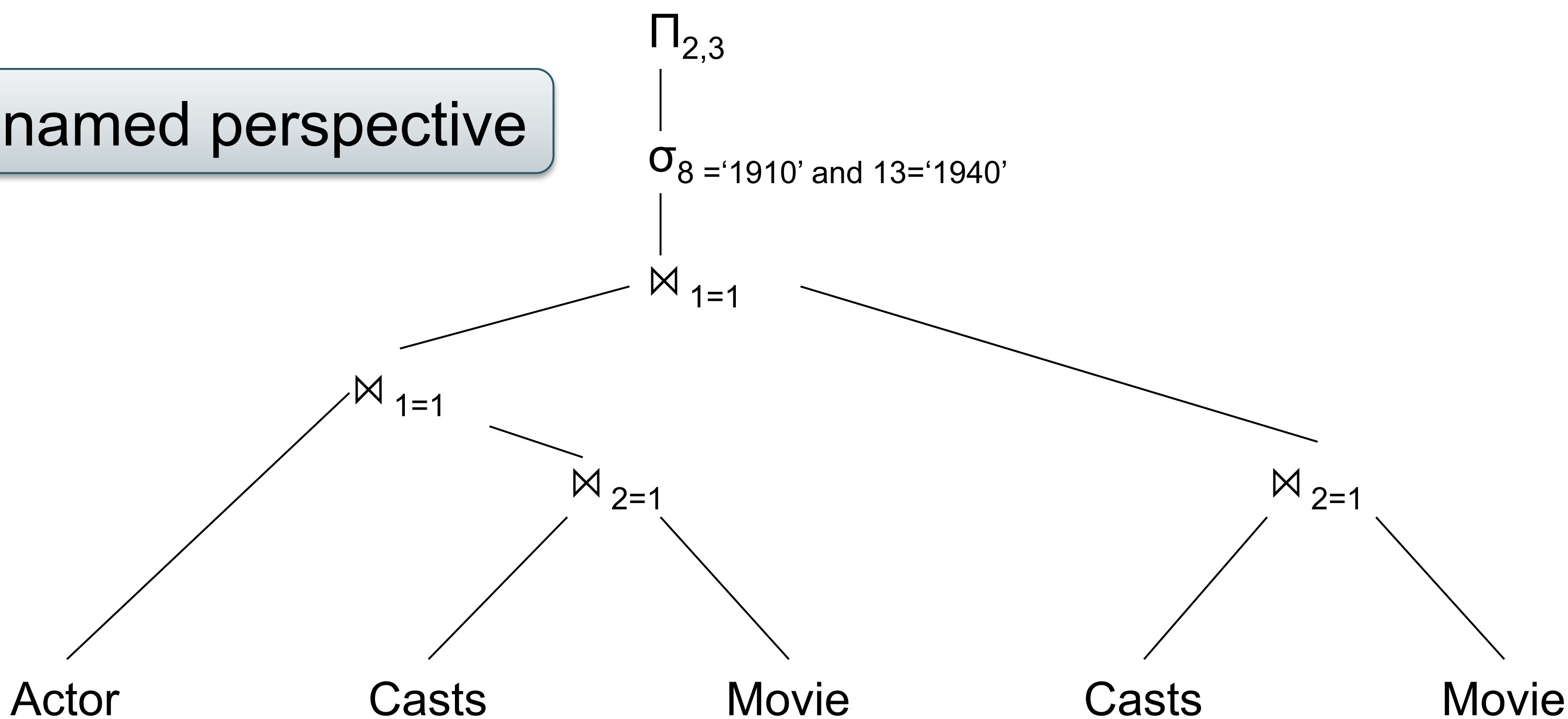
Note how we renamed year to year1, year2

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

1. Relational Algebra

```
Q: SELECT DISTINCT a.fname, a.lname
   FROM   Actor a, Casts c1, Movie m1, Casts c2, Movie m2
   WHERE  a.id = c1.pid       AND c1.mid = m1.id
      AND a.id = c2.pid       AND c2.mid = m2.id
      AND m1.year = 1910      AND m2.year = 1940;
```

Unnamed perspective



2. Datalog

- Very friendly notation for queries
- Designed for *recursive* queries in the 80s
- Today: a couple of commercial products, e.g. LogicBlox

- In class
  - *recursion-free* datalog with negation (next)
  - *recursive datalog*, (in the "Theory" part)

2. Datalog

How to try out datalog quickly:

- Download DLV from http://www.dbai.tuwien.ac.at/proj/dlv/
- Run DLV on this file:

```
parent(william, john).
parent(john, james).
parent(james, bill).
parent(sue, bill).
parent(james, carol).
parent(sue, carol).

male(john).
male(james).
female(sue).
male(bill).
female(carol).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
father(X, Y) :- parent(X, Y), male(X).
mother(X, Y) :- parent(X, Y), female(X).
brother(X, Y) :- parent(P, X), parent(P, Y), male(X), X != Y.
sister(X, Y)  :- parent(P, X), parent(P, Y), female(X), X != Y.
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

2. Datalog: Facts and Rules

Facts = tuples in the database

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

Rules = queries

```
Q1(y) :-  Movie(x,y,z), z='1940'.
```

Find Movies made in 1940

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

2. Datalog: Facts and Rules

Facts = tuples in the database

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

Rules = queries

```
Q1(y) :-  Movie(x,y,z), z='1940'.
```

```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x),
               Movie(x,y,'1940').
```

Find Actors who acted in Movies made in 1940

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

2. Datalog: Facts and Rules

Facts = tuples in the database

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

Rules = queries

```
Q1(y) :-  Movie(x,y,z), z='1940'.
```

```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x),
             Movie(x,y,'1940').
```

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
                          Casts(z,x2), Movie(x2,y2,1940)
```

Find Actors who acted in a Movie in 1940 and in one in 1910

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

2. Datalog: Facts and Rules

Facts = tuples in the database

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

Rules = queries

```
Q1(y) :-  Movie(x,y,z), z='1940'.
```

```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x),
              Movie(x,y,'1940').
```

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
                          Casts(z,x2), Movie(x2,y2,1940)
```

Extensional Database Predicates = EDB = Actor, Casts, Movie

Intensional Database Predicates = IDB = Q1, Q2, Q3

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

2. Datalog: Terminology

$$\overbrace{\text{Q2(f, l)}}^{\text{head}} \text{ :- } \overbrace{\overbrace{\text{Actor(z,f,l)}}^{\text{atom}}\text{, }\overbrace{\text{Casts(z,x)}}^{\text{atom}}\text{, }\overbrace{\text{Movie(x,y,'1940')}}^{\text{atom}}}^{\text{body}}\text{.}$$

f, l = head variables
x,y,z = existential variables

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

2. Datalog program

Find all actors with Bacon number ≤ 2

```
B0(x) :- Actor(x,'Kevin', 'Bacon')
B1(x) :- Actor(x,f,l), Casts(x,z), Casts(y,z), B0(y)
B2(x) :- Actor(x,f,l), Casts(x,z), Casts(y,z), B1(y)
Q4(x) :- B1(x)
Q4(x) :- B2(x)
```

Note: Q4 is the *union* of B1 and B2

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

2. Datalog with negation

Find all actors with Bacon number ≥ 2

```
B0(x) :- Actor(x,'Kevin', 'Bacon')
B1(x) :- Actor(x,f,l), Casts(x,z), Casts(y,z), B0(y)
Q6(x) :- Actor(x,f,l), not B1(x), not B0(x)
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

2. Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

U1(x,y) :- Movie(x,z,1994), y>1910

U2(x)   :- Movie(x,z,1994), not Casts(u,x)

A datalog rule is *safe* if every variable appears in some positive relational atom

2. Datalog v.s. SQL

- Non-recursive datalog with negation is a cleaned-up, core of SQL

- You should be able to translate easily between non-recursive datalog with negation and SQL

3. Relational Calculus

- *Predicate calculus*, or *first order logic*
- The most expressive formalism for queries: easy to write complex queries

- TRC = Tuple RC = named perspective
- DRC = Domain RC = unnamed perspective

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

3. Relational Calculus

Predicate P:

$$P ::= \text{atom} \mid P \land P \mid P \lor P \mid P \Rightarrow P \mid \text{not}(P) \mid \forall x.P \mid \exists x.P$$

Query Q:

$$Q(x1, \ldots, xk) = P$$

Example: find the first/last names of actors who acted in 1940

$$Q(f,l) = \exists x.\ \exists y.\ \exists z.\ (\text{Actor}(z,f,l) \land \text{Casts}(z,x) \land \text{Movie}(x,y,1940))$$

What does this query return ?

$$Q(f,l) = \exists z.\ (\text{Actor}(z,f,l) \land \forall x.(\text{Casts}(z,x) \Rightarrow \exists y.\text{Movie}(x,y,1940)))$$

3. Relational Calculus: Example

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Find drinkers that frequent <u>some</u> bar that serves <u>some</u> beer they like.

$$Q(x) = \exists y.\ \exists z.\ \text{Frequents}(x, y) \wedge \text{Serves}(y,z) \wedge \text{Likes}(x,z)$$

3. Relational Calculus: Example

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Find drinkers that frequent <u>some</u> bar that serves <u>some</u> beer they like.

$$Q(x) = \exists y.\ \exists z.\ \text{Frequents}(x, y) \land \text{Serves}(y,z) \land \text{Likes}(x,z)$$

Find drinkers that frequent <u>only</u> bars that serves <u>some</u> beer they like.

$$Q(x) = \forall y.\ \text{Frequents}(x, y) \Rightarrow (\exists z.\ \text{Serves}(y,z) \land \text{Likes}(x,z))$$

3. Relational Calculus: Example

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Find drinkers that frequent <u>some</u> bar that serves <u>some</u> beer they like.

$$Q(x) = \exists y.\ \exists z.\ \text{Frequents}(x, y) \wedge \text{Serves}(y,z) \wedge \text{Likes}(x,z)$$

Find drinkers that frequent <u>only</u> bars that serves <u>some</u> beer they like.

$$Q(x) = \forall y.\ \text{Frequents}(x, y) \Rightarrow (\exists z.\ \text{Serves}(y,z) \wedge \text{Likes}(x,z))$$

Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they like.

$$Q(x) = \exists y.\ \text{Frequents}(x, y) \wedge \forall z.(\text{Serves}(y,z) \Rightarrow \text{Likes}(x,z))$$

3. Relational Calculus: Example

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Find drinkers that frequent <u>some</u> bar that serves <u>some</u> beer they like.

$$\boxed{Q(x) = \exists y.\ \exists z.\ \text{Frequents}(x, y) \wedge \text{Serves}(y,z) \wedge \text{Likes}(x,z)}$$

Find drinkers that frequent <u>only</u> bars that serves <u>some</u> beer they like.

$$\boxed{Q(x) = \forall y.\ \text{Frequents}(x, y) \Rightarrow (\exists z.\ \text{Serves}(y,z) \wedge \text{Likes}(x,z))}$$

Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they like.

$$\boxed{Q(x) = \exists y.\ \text{Frequents}(x, y) \wedge \forall z.(\text{Serves}(y,z) \Rightarrow \text{Likes}(x,z))}$$

Find drinkers that frequent <u>only</u> bars that serves <u>only</u> beer they like.

$$\boxed{Q(x) = \forall y.\ \text{Frequents}(x, y) \Rightarrow \forall z.(\text{Serves}(y,z) \Rightarrow \text{Likes}(x,z))}$$

3. Domain Independent Relational Calculus

- As in datalog, one can write “unsafe” RC queries; they are also called *domain dependent*
- Checking whether a query is safe is undecidable. ☹

- Lesson: make sure your RC queries are domain independent

3. Relational Calculus

How to write a complex SQL query:

- Write it in RC
- Translate RC to datalog (see next)
- Translate datalog to SQL

Take shortcuts when you know what you're doing

3. From RC to Non-recursive Datalog w/ negation

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$Q(x) = \exists y.\ \text{Likes}(x, y) \wedge \forall z.(\text{Serves}(z,y) \Rightarrow \text{Frequents}(x,z))$$

3. From RC to Non-recursive Datalog w/ negation

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(Serves(z,y) \Rightarrow Frequents(x,z))$$

Step 1: Replace ∀ with ∃ using de Morgan's Laws

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg\exists z.(Serves(z,y) \wedge \neg Frequents(x,z))$$

3. From RC to Non-recursive Datalog w/ negation

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(Serves(z,y) \Rightarrow Frequents(x,z))$$

Step 1: Replace ∀ with ∃ using de Morgan's Laws

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg\exists z.(Serves(z,y) \wedge \neg Frequents(x,z))$$

Step 2: Make all subqueries domain independent

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg\exists z.(Likes(x,y) \wedge Serves(z,y) \wedge \neg Frequents(x,z))$$

3. From RC to Non-recursive Datalog w/ negation

$$Q(x) = \exists y.\ \text{Likes}(x, y) \wedge \neg \underbrace{\exists z.(\text{Likes}(x,y) \wedge \text{Serves}(z,y) \wedge \neg\text{Frequents}(x,z))}_{H(x,y)}$$

Step 3: Create a datalog rule for each subexpression;
(shortcut: only for "important" subexpressions)

```
H(x,y)   :- Likes(x,y),Serves(y,z), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

3. From RC to Non-recursive Datalog w/ negation

```
H(x,y)   :- Likes(x,y),Serves(y,z), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Step 4: Write it in SQL

```sql
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Likes L2, Serves S
   WHERE L2.drinker=L.drinker and L2.beer=L.beer
         and L2.beer=S.beer
         and not exists (SELECT * FROM Frequents F
                         WHERE F.drinker=L2.drinker
                           and F.bar=S.bar))
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

3. From RC to Non-recursive Datalog w/ negation

```
H(x,y)   :- ~~Likes(x,y),~~Serves(y,z), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Unsafe rule

Improve the SQL query by using an unsafe datalog rule

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Serves S
   WHERE L.beer=S.beer
         and not exists (SELECT * FROM Frequents F
                         WHERE F.drinker=L.drinker
                               and F.bar=S.bar))
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Summary of Translation

- RC → recursion-free datalog w/ negation
  - Subtle: as we saw; more details in the paper
- Recursion-free datalog w/ negation → RA
- RA → RC

Theorem: RA, non-recursive datalog w/ negation, and RC, express exactly the same sets of queries: RELATIONAL QUERIES