

DATABASE SYSTEMS

Copyright © 2009 by Leonid Libkin

These slides are free to download for students and instructors. If you use them for teaching a course, you can only distribute them to students *free of charge*. These slides cannot be posted on other web sites without permission of the author.

The relational model

- Data is organized in relations (tables)
- Relational database schema:
 - set of table names
 - list of attributes for each table
- Tables are specified as: <table name>:<list of attributes>
- Examples:
 - Account: number, branch, customerId
 - Movie: title, director, actor
 - Schedule: theater, title
- Attributes within a table have different names
- Tables have different names

Example: relational database

Movie	title	director	actor
	Shining	Kubrick	Nicholson
	Player	Altman	Robbins
	Chinatown	Polanski	Nicholson
	Chinatown	Polanski	Polanski
	Repulsion	Polanski	Deneuve

Schedule	theater	title
	Le Champo	Shining
	Le Champo	Chinatown
	Le Champo	Player
	Odéon	Chinatown
	Odéon	Repulsion

Examples of queries

- Find titles of current movies:

answer	title
	Shining
	Player
	Chinatown
	Repulsion

- Find theaters showing movies directed by Polanski:

answer	theater
	Le Champo
	Odéon

- Find theaters showing movies featuring Nicholson:

answer	theater
	Le Champo
	Odéon

- Find all directors who directed themselves:

answer	director
	Polanski

- Find directors whose movies are playing in all theaters:

answer	director
	Polanski

- Find theaters that only show movies featuring Nicholson:

answer	theater

but if Le Champo stops showing 'Player', the answer to the query becomes:

answer	theater
	Le Champo

Query results

- They are tables constructed from tables in the database

How to ask a query?

- Query languages
 - Commercial: SQL
 - Theoretical: Relational calculus, algebra, datalog etc

Declarative vs Procedural

- In our queries, we ask **what** we want to see in the output.
- But we do not say **how** we want to get this output.
- Thus, query languages are **declarative**: they specify what is needed in the output, but do not say how to get it.
- Database system figures out **how** to get the result, and gives it to the user.
- Database system operates internally with different, **procedural** languages, which specify how to get the result.

Declarative vs Procedural: example

Declarative:

$\{ \text{title} \mid (\text{title}, \text{director}, \text{actor}) \in \text{movies} \}$

Procedural:

for each tuple $T=(t,d,a)$ in relation movies do

 output t

end

Declarative vs Procedural: another example

Declarative:

```
{ theater | (title, director, actor) ∈ movies,  
  (theater, title) ∈ schedule,  
  actor='Nicholson' }
```

Procedural:

```
for each tuple T1=(t1,d,a) in relation movies do  
  for each tuple T2=(th,t2) in relation schedule do  
    if t1 = t2 and a='Nicholson' then output th  
  end  
end
```

Declarative vs Procedural

- Theoretical languages:
 - Declarative: relational calculus, rule-based queries
 - Procedural: relational algebra
- Practical languages: mix of both, but mostly one uses declarative features.

What's next?

We'll do examples of queries in various query languages.

Examples of queries

- Find titles of current movies:

$\text{answer}(tl) :- \text{movies}(tl, \text{dir}, \text{act})$

- That is, while $(tl, \text{dir}, \text{act})$ ranges over relation `movies`, output `tl` (the title attribute)
- We formulate queries as *rules*, that say when certain elements belong to the answer.
- Queries like this one are called *conjunctive queries*; we'll see later why.

Next example

- Find theaters showing movies directed by Polanski:

$\text{answer}(\text{th}) \text{ :- movies}(\text{tl}, \text{'Polanski'}, \text{act}), \text{schedule}(\text{th}, \text{tl})$

- While $(\text{tl}, \text{dir}, \text{act})$ range over tuples in `movies`, check if `dir` is 'Polanski'; if not, go to the next tuple, if yes, look at all tuples (th, tl) in `schedule` corresponding to the title `tl` in relation `movies`, and output `th`.

This is the most common type of queries one asks.

- Find directors who acted in their own movies:

$\text{answer}(\text{dir}) \text{ :- movies}(\text{tl}, \text{dir}, \text{act}), \text{dir}=\text{act}$

- While $(\text{tl}, \text{dir}, \text{act})$ ranges over tuples in movies, check if dir is the same as act , and output it if that is the case.

A more complicated example

- Find directors whose movies are playing in all theaters.
- “All” is often problematic: one needs *universal quantifier* \forall .
- We use notation from mathematical logic:
- $\{ \text{dir} \mid \forall (\text{th}, \text{tl}) \in \text{schedule} \\ \exists (\text{tl}', \text{act}): (\text{tl}', \text{dir}, \text{act}) \in \text{movies} \wedge (\text{th}, \text{tl}') \in \text{schedule} \}$
- That is, to see if director *dir* is in the answer, for each theater name *th*, check that there exists a tuple (*tl'*, *dir*, *act*) in *movies*, and a tuple (*th*, *tl'*) in *schedule*

Reminder:

- \forall means “for all”, \exists means “exists”
- \wedge is conjunction (logical AND)

SQL

- Structured Query Language
- Developed originally at IBM in the late 70s
- First standard: SQL-86
- Second standard: SQL-92
- Latest standard: SQL-99, or SQL3, well over 1,000 pages
- "The nice thing about standards is that you have so many to choose from." – Andrew S. Tanenbaum.
- De-facto standard of the relational database world – replaced all other languages.

Examples of SQL queries

- Find titles of current movies

```
SELECT Title  
FROM Movies
```

- SELECT lists attributes that go into the output of a query
- FROM lists input relations

More examples

- Find theaters showing movies directed by Polanski:

```
SELECT Schedule.Theater
FROM Schedule, Movies
WHERE Movies.Title = Schedule.Title
      AND Movies.Director='Polanski'
```

Differences:

- SELECT now specifies which relation the attributes came from – because we use more than one.
- FROM lists two relations
- WHERE specifies the *condition* for selecting a tuple.

Joining relations

- WHERE allows us to join together several relations
- Consider a query: list directors, and theaters in which their movies are playing
- Conjunctive query:

$\text{answer}(\text{dir}, \text{th}) \text{ :- schedule}(\text{th}, \text{tl}), \text{movies}(\text{tl}, \text{dir}, \text{act})$

- SQL query:

```
SELECT Movies.Director, Schedule.Theater
FROM Movies, Schedule
WHERE Movies.Title = Schedule.Title
```

Joining relations cont'd

- `SELECT Movies.Director, Schedule.Theater`
`FROM Movies, Schedule`
`WHERE Movies.Title = Schedule.Title`
- Semantics: nested loops over relations listed in FROM

for each tuple (Title1, Director, Actor) in Movies do
 for each tuple (Theater, Title2) in Schedule do
 if Title1=Title2 then output (Director, Theater)
 end
end
- This operation is called **join**. It is one of the most fundamental operations in database queries.

We will see many examples throughout the course.

Procedural Language: Relational algebra

- We start with a subset of relational algebra that suffices to capture queries defined by simple rules, and by SQL SELECT-FROM-WHERE statements.
- The subset has three operations:
 - Projection π
 - Selection σ
 - Cartesian Product \times
- Sometimes we also use renaming ρ but it can be avoided.

Projection

- Chooses some attributes in a relation
- $\pi_{A_1, \dots, A_n}(R)$: only leaves attributes A_1, \dots, A_n in relation R .
- Example:

$$\pi_{\text{title,director}} \left(\begin{array}{ccc} \text{title} & \text{director} & \text{actor} \\ \hline \text{Shining} & \text{Kubrick} & \text{Nicholson} \\ \text{Player} & \text{Altman} & \text{Robbins} \\ \text{Chinatown} & \text{Polanski} & \text{Nicholson} \\ \text{Chinatown} & \text{Polanski} & \text{Polanski} \\ \text{Repulsion} & \text{Polanski} & \text{Deneuve} \end{array} \right) = \begin{array}{cc} \text{title} & \text{director} \\ \hline \text{Shining} & \text{Kubrick} \\ \text{Player} & \text{Altman} \\ \text{Chinatown} & \text{Polanski} \\ \text{Repulsion} & \text{Polanski} \end{array}$$

- Provides the user with a *view* of data by hiding some attributes

Selection

- Chooses tuples that satisfy some condition
- $\sigma_c(R)$: only leaves tuples t for which $c(t)$ is true
- Conditions: conjunctions of
 - $R.A = R.A'$ – two attributes are equal
 - $R.A = constant$ – the value of an attribute is a given constant
 - Same as above but with \neq instead of $=$
- Examples:
 - Movies.Actor=Movies.Director
 - Movies.Actor \neq 'Nicholson'
 - Movies.Actor=Movies.Director \wedge Movies.Actor='Nicholson'
- Provides the user with a *view* of data by hiding tuples that do not satisfy the condition the user wants.

Selection: Example

$$\sigma_{\text{actor}=\text{director} \wedge \text{director}='Polanski'}$$

title	director	actor
Shining	Kubrick	Nicholson
Player	Altman	Robbins
Chinatown	Polanski	Nicholson
Chinatown	Polanski	Polanski
Repulsion	Polanski	Deneuve

$$=$$

title	director	actor
Chinatown	Polanski	Polanski

Combining selection and projection

- Find directors who acted in their movies
- $\text{answer}(\text{dir}) := \text{movies}(\text{tl}, \text{dir}, \text{act}), \text{act} = \text{dir}$

- ```
SELECT Director
FROM Movies
WHERE Director=Actor
```

- Relational algebra query:

$$Q = \pi_{\text{director}} (\sigma_{\text{director}=\text{actor}} (\text{Movies}))$$

- $\sigma_{\text{director}=\text{actor}} (\text{Movies})$  gives us 

| title     | director | actor    |
|-----------|----------|----------|
| Chinatown | Polanski | Polanski |

- Thus  $\pi_{\text{director}} (\sigma_{\text{director}=\text{actor}} (\text{Movies}))$  gives us 

| director |
|----------|
| Polanski |

## Combining selection and projection cont'd

- There could be more than one way to write selection-projection queries
- Example: find movies and directors excluding Polanski's movies
- $\text{answer}(\text{tl}, \text{dir}) :- \text{movies}(\text{tl}, \text{dir}, \text{act}), \text{dir} \neq \text{'Polanski'}$
- Relational algebra query:

$$Q_1 = \sigma_{\text{director} \neq \text{'Polanski'}} (\pi_{\text{title}, \text{director}}(\text{Movies}))$$

- An equivalent relational algebra query

$$Q_2 = \pi_{\text{title}, \text{director}} (\sigma_{\text{director} \neq \text{'Polanski'}} (\text{Movies}))$$

- The same declarative query can be translated into more than one procedural query

## Combining selection and projection cont'd

- Are  $Q_1$  and  $Q_2$  the same?
- They are the same semantically, as they produce the same result.
- But differ in terms of their *efficiency*.
- $Q_1$  scans Movies, projects out two attributes, and scans the the result again.
- $Q_2$  scans movies, selects some tuples, and then only scans *selected tuples*
- Thus, it is likely that  $Q_2$  is more efficient
- Procedural languages can be *optimized*: there are semantically equivalent ways to write the same query, and some of those ways are more efficient

## Cartesian Product

- Puts together two relations
- $R_1 \times R_2$  puts together each tuple  $t_1$  of  $R_1$  and each tuple  $t_2$  of  $R_2$
- Example:

$$\begin{array}{c|cc} R_1 & A & B \\ \hline & a_1 & b_1 \\ & a_2 & b_2 \end{array} \times \begin{array}{c|cc} R_2 & A & C \\ \hline & a_1 & c_1 \\ & a_2 & c_2 \\ & a_3 & c_3 \end{array} = \begin{array}{cccc} R_1.A & R_1.B & R_2.A & R_2.C \\ \hline a_1 & b_1 & a_1 & c_1 \\ a_1 & b_1 & a_2 & c_2 \\ a_1 & b_1 & a_3 & c_3 \\ a_2 & b_2 & a_1 & c_1 \\ a_2 & b_2 & a_2 & c_2 \\ a_2 & b_2 & a_3 & c_3 \end{array}$$

- We renamed attributes to include the name of the relation: in the resulting table, all attributes must have different names.

## Cartesian Product cont'd

- If  $R_1$  has  $n$  tuples and  $R_2$  has  $m$  tuples, then  $R_1 \times R_2$  has  $n \times m$  tuples
- This is an expensive operation: if  $R$  and  $S$  each have 1,000 tuples (small relations),  $R \times S$  has 1,000,000 tuples (quite large)
- Query processing algorithms try to avoid building products – instead they attempt to build only subsets which contain relevant information.

## Cartesian Product: Example

- Find theaters playing movies directed by Polanski
- $\text{answer}(\text{th}) \text{ :- movies}(\text{tl}, \text{dir}, \text{act}), \text{schedule}(\text{th}, \text{tl}), \text{dir} = \text{'Polanski'}$
- Step 1: Let  $R_1 = \text{Movies} \times \text{Schedule}$
- We don't need all tuples, only those in which titles are the same, so:
- Step 2: Let  $R_2 = \sigma_{\text{cond}}(R_1)$  where  $\text{cond}$  is  $\text{Movies.title} = \text{Schedule.title}$
- We are only interested in movies directed by Polanski, so  
 $R_3 = \sigma_{\text{director} = \text{'Polanski'}}(R_2)$
- In the output, we only want theaters, so finally  
 $\text{Answer} = \pi_{\text{theater}}(R_3)$
- Summing up, the answer is

$$\pi_{\text{theater}}(\sigma_{\text{director} = \text{'Polanski'}}(\sigma_{\text{Movies.title} = \text{Schedule.title}}(\text{Movies} \times \text{Schedule})))$$

## Cartesian Product: Example cont'd

- Several selections can be combined into one:
- $\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_1 \wedge c_2}(R)$
- So the answer to the query is

$$\pi_{\text{theater}}(\sigma_{\text{director}='Polanski' \wedge \text{Movies.title}=\text{Schedule.title}}(\text{Movies} \times \text{Schedule}))$$

## SQL and relational algebra

- We have to translate declarative languages into procedural languages

- Idea:

SELECT is projection  $\pi$

FROM is Cartesian product  $\times$

WHERE is selection  $\sigma$

- A simple case: only one relation in FROM

SELECT  $A, B, \dots$

FROM  $R$

WHERE condition  $c$

is translated into

$$\pi_{A,B,\dots}(\sigma_c(R))$$

## Translating declarative queries into relational algebra

- Find titles of all movies
- $\text{answer}(tl) \text{ :- movies}(tl,dir,act)$
- ```
SELECT Title  
FROM Movies
```
- This is simply projection:

$$\pi_{\text{title}}(\text{Movies})$$

Translation Examples

- Find theaters showing movies directed by Polanski:
- ```
SELECT Schedule.Theater
FROM Schedule, Movies
WHERE Movies.Title = Schedule.Title
 AND Movies.Director='Polanski'
```
- First, translate into a rule:  

```
answer(th) :- schedule(th,tl), movies(tl,'Polanski',act)
```
- Second, change into a rule such that:  
    constants appear only in conditions  
    no two variables are the same
- This gives us:  

```
answer(th) :- schedule(th,tl), movies(tl',dir,act), dir = 'Polanski', tl=tl'
```

## Translation Examples cont'd

- $\text{answer}(\text{th}) \text{ :- schedule}(\text{th}, \text{tl}), \text{ movies}(\text{tl}', \text{dir}, \text{act}), \text{ dir} = \text{'Polanski'}, \text{ tl} = \text{tl}'$

Two relations  $\implies$  Cartesian product

Conditions  $\implies$  selection

Subset of attributes in the answer  $\implies$  projection

- Step 1:  $R_1 = \text{Schedule} \times \text{Movies}$

- Step 2: Make sure we talk about the same movie:

$$R_2 = \sigma_{\text{Schedule.title}=\text{Movies.title}}(R_1)$$

- Step 3: We are only interested in Polanski's movies:

$$R_3 = \sigma_{\text{Movies.director}=\text{Polanski}}(R_2)$$

- Step 4: we need only theaters in the output

$$\text{answer} = \pi_{\text{schedule.theater}}(R_3)$$

## Translation Examples cont'd

Summing up, the answer is:

$$\pi_{\text{schedule.theater}} \left( \sigma_{\text{Movies.director=Polanski}} \left( \sigma_{\text{Schedule.title=Movies.title}} (\text{Schedule} \times \text{Movies}) \right) \right)$$

or, using the rule  $\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_1 \wedge c_2}(R)$ :

$$\pi_{\text{schedule.theater}} \left( \sigma_{\text{Movies.director=Polanski} \wedge \text{Schedule.title=Movies.title}} (\text{Schedule} \times \text{Movies}) \right)$$

## Formal translation: SQL to rule-based queries

```
SELECT attribute list $\langle R_i.A_j \rangle$
FROM R_1, \dots, R_n
WHERE condition c
```

is translated into:

$$\text{answer}(\langle R_i.A_j \rangle) \quad :- \quad \begin{array}{l} R_1(\langle \text{attributes} \rangle), \\ \dots, \\ R_n(\langle \text{attributes} \rangle), \\ c \end{array}$$

## Rules into Relational algebra

- How are rules translated into algebra?

$$\text{answer}(a_1, \dots, a_k) \text{ :- } R_1(\vec{A}_1), \dots, R_n(\vec{A}_n), \text{ conditions}$$

- First, make sure no two attributes are the same: if we have  $R_i(\dots, A, \dots)$  and  $R_j(\dots, A, \dots)$ , turn them into  $R_i(\dots, A', \dots)$  and  $R_j(\dots, A'', \dots)$ , and add  $A' = A''$  to the conditions.
- For example,  $\text{answer}(\text{th}, \text{dir}) \text{ :- } \text{movies}(\text{tl}, \text{dir}, \text{act}), \text{schedule}(\text{th}, \text{tl})$  is rewritten to  
 $\text{answer}(\text{th}, \text{dir}) \text{ :- } \text{movies}(\text{tl}', \text{dir}, \text{act}), \text{schedule}(\text{th}, \text{tl}''), \text{tl}' = \text{tl}''$
- Such rules are translated into

$$\pi_{a_1, \dots, a_k}(\sigma_{\text{conditions}}(R_1 \times \dots \times R_n))$$

## Putting it together: SQL into relational algebra

- Combining translations:

SQL into rule-based queries and rule-based into relational algebra we have the following SQL to relational algebra translation:

- 

```
SELECT attribute list $\langle R_i.A_j \rangle$
FROM R_1, \dots, R_n
WHERE condition c
```

is translated into

$$\pi_{\langle R_i.A_j \rangle}(\sigma_c(R_1 \times \dots \times R_n))$$

## Another example

- Find theaters showing movies featuring Nicholson.
- ```
SELECT Schedule.Theater
FROM Schedule, Movies
WHERE Movies.Title = Schedule.Title
      AND Movies.Actor='Nicholson'
```
- Translate into a rule:
answer(th) :- movies(tl, dir, 'Nicholson'), schedule(th, tl)
- Modify the rule:
answer(th) :- movies(tl, dir, act), schedule(th, tl'), tl=tl', act='Nicholson'

Another example cont'd

- $\text{answer}(\text{th}) :- \text{movies}(\text{tl}, \text{dir}, \text{act}), \text{schedule}(\text{th}, \text{tl}'), \text{tl}=\text{tl}', \text{act}=\text{'Nicholson'}$
- Step 1: $R_1 = \text{Schedule} \times \text{Movies}$
- Step 2: Make sure we talk about the same movie:

$$R_2 = \sigma_{\text{Schedule.title}=\text{Movies.title}}(R_1)$$

- Step 3: We are only interested in movies with Nicholson:

$$R_3 = \sigma_{\text{Movies.actor}=\text{Nicholson}}(R_2)$$

- Step 4: we need only theaters in the output

$$\text{answer} = \pi_{\text{schedule.theater}}(R_3)$$

Summing up:

$$\pi_{\text{schedule.theater}}\left(\sigma_{\text{Movies.actor}=\text{Nicholson}} \wedge \text{Schedule.title}=\text{Movies.title}(\text{Schedule} \times \text{Movies})\right)$$

Natural Join

- We have seen the following common steps in the last two queries:

Step 1: $R_1 = \text{Schedule} \times \text{Movies}$

Step 2: Make sure we talk about the same movie:

$$R_2 = \sigma_{\text{Schedule.title}=\text{Movies.title}}(R_1)$$

- Attributes of R_2 :
Schedule.Theater, Schedule.Title,
Movies.Title, Movies.Director, Movies.Actor
- But one of the title attributes is redundant:
Movies.Title and Schedule.Title are always the same in R_2 .
- Thus, we can reduce R_2 to a simple relation with the attributes:
Schedule.Theater, Title, Movies.Director, Movies.Actor
- This is the **natural join** $\text{Schedule} \bowtie \text{Movies}$

Natural join: example

title	director	actor		theater	title	
Shining	Kubrick	Nicholson		Le Champo	Shining	
Player	Altman	Robbins	⋈	Le Champo	Chinatown	=
Chinatown	Polanski	Nicholson		Le Champo	Player	
Chinatown	Polanski	Polanski		Odéon	Chinatown	
Repulsion	Polanski	Deneuve		Odéon	Repulsion	

	title	director	actor	theater
	Shining	Kubrick	Nicholson	Le Champo
	Player	Altman	Robbins	Le Champo
=	Chinatown	Polanski	Nicholson	Le Champo
	Chinatown	Polanski	Nicholson	Odéon
	Chinatown	Polanski	Polanski	Le Champo
	Chinatown	Polanski	Polanski	Odéon
	Repulsion	Polanski	Deneuve	Odéon

Join cont'd

- Join is not a new operation of relational algebra
- It is **definable** with π, σ, \times
- Suppose R is a relation with attributes $A_1, \dots, A_n, B_1, \dots, B_k$
- S is a relation with attributes $A_1, \dots, A_n, C_1, \dots, C_m$
- $R \bowtie S$ has attributes $A_1, \dots, A_n, B_1, \dots, B_k, C_1, \dots, C_m$

$$\begin{aligned} & R \bowtie S \\ = & \pi_{A_1, \dots, A_n, B_1, \dots, B_k, C_1, \dots, C_m} \left(\sigma_{R.A_1=S.A_1 \wedge \dots \wedge R.A_n=S.A_n} (R \times S) \right) \end{aligned}$$

Properties of join

- Commutative: $R \bowtie S = S \bowtie R$
- Associative: $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
- Hence we can write $R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$

Commutativity and Associativity of Joins

R	employee	department
	Jones	D1
	Brown	D2
	Smith	D3

S	department	office
	D1	USA
	D2	UK

T	office	head
	USA	Andrews
	UK	Morrison

$R \bowtie S \bowtie T$	employee	department	office	head
	Jones	D1	USA	Andrews
	Brown	D2	UK	Morrison

Select-Project-Join (SPJ) queries

- These are the most common queries
- Simple rules, or simple SELECT-FROM-WHERE queries.
- Find theaters showing movies directed by Polanski:
- $\text{answer}(\text{th}) \text{ :- schedule}(\text{th}, \text{tl}), \text{movies}(\text{tl}, \text{'Polanski'}, \text{act})$
- As SPJ query:

$$\pi_{\text{theater}}(\sigma_{\text{director}=\text{'Polanski'}}(\text{Movies} \bowtie \text{Schedule}))$$

- What is simpler compared to earlier version?

$$\pi_{\text{schedule.theater}}(\sigma_{\text{Movies.director}=\text{Polanski}}(\sigma_{\text{Schedule.title}=\text{Movies.title}}(\text{Schedule} \times \text{Movies})))$$

- Selection $\text{Schedule.title}=\text{Movies.title}$ is eliminated; it is implied by the join.

Properties of relational algebra operators

Sizes of outputs

- Projection: $\text{size}(\pi(R)) \leq \text{size}(R)$
- But sometimes $\text{size}(\pi(R)) < \text{size}(R)$
- This happens if some attribute values are the same

$$\pi_A \left(\begin{array}{cc} A & B \\ a & b1 \\ a & b2 \end{array} \right) = \begin{array}{c} A \\ a \end{array}$$

- Selection: $0 \leq \text{size}(\sigma(R)) \leq \text{size}(R)$
- Depends on how many tuples satisfy the condition.

Sizes of joins and cartesian products

- $\text{size}(R \times S) = \text{size}(R) \times \text{size}(S)$, but:
- $0 \leq \text{size}(R \bowtie S) \leq \text{size}(R) \times \text{size}(S)$
- Dangling tuples that do not participate in the join:

R	employee	department	\bowtie	S	department	office
	Jones	D1			D1	USA
	Brown	D2			D2	UK
	Smith	D3				

=	$R \bowtie S$	employee	department	office
		Jones	D1	USA
		Brown	D2	UK

- (Smith, D3) is not joined with any tuple in S : S has no information about department D3.

Empty joins

- Joins could be empty:

R	employee	department		S'	department	office
	Jones	D1	\bowtie		D4	France
	Brown	D2			D5	Italy
	Smith	D3				

=

$R \bowtie S'$	employee	department	office

Translating SPJ queries back into rules and SQL

- $Q = \pi_{\vec{A}}(\sigma_c(R \bowtie S))$
- Let B_1, \dots, B_m be the common attributes in R and S
- Equivalent SQL statement:

- ```
SELECT \vec{A}
FROM R, S
WHERE $c, R.B_1 = S.B_1$ AND ... AND $R.B_m = S.B_m$
```

- Equivalent rule query:

answer( $\vec{A}$ ) :-  $R(\langle \text{attributes of } R \rangle), S(\langle \text{attributes of } S \rangle),$   
 $R.B_1 = S.B_1, \dots, R.B_m = S.B_m, c$

## SPJ to SQL: Example

- Find directors of currently playing movies featuring Ford:
- $\pi_{\text{director}}(\sigma_{\text{actor}='Ford'}(\text{Movies} \bowtie \text{Schedule}))$
- In SQL:

```
SELECT Movies.director
FROM Movies, Schedule
WHERE Movies.title=Schedule.title AND Movies.actor='Ford'
```

## What we've seen so far

- Simple queries given by SQL `SELECT-FROM-WHERE`
- Same queries are defined by rules
- They are also the same queries as those definable by  $\pi, \sigma, \times$  in relational algebra

## Saving space

- We don't want to repeat relation names many times
- SQL lets you use temporary names for relations
- ```
SELECT S.Theater  
FROM Schedule S, Movies M  
WHERE S.Title=M.Title AND M.Director='Polanski'
```
- Using a variable after relation name indicates that the relation is temporarily renamed.

Nested queries: simple example

- So far in the WHERE clause we used comparisons of attributes.
- In general, a WHERE clause could contain *another query*, and test some relationship between an attribute and the result of that query.
- We call queries like this *nested*, as they use *subqueries*
- Example: Find theaters showing Polanski's movies

```
SELECT Schedule.Theater
FROM Schedule
WHERE Schedule.Title IN
      (SELECT Movies.Title
       FROM Movies
       WHERE Movies.Director='Polanski')
```

Nested queries: comparison

```
SELECT S.Theater
FROM Schedule S
WHERE S.Title IN
      (SELECT M.Title
       FROM Movies M
       WHERE M.Director='Polanski')
```

```
SELECT S.Theater
FROM Schedule S, Movies M
WHERE S.Title=M.Title
      AND M.Director='Polanski'
```

- These express the same query
- On the left, each subquery refers to one relation
- The real advantage of nesting is that one can use more complex predicates than IN.

Disjunction in queries

- Find actors who played in movies directed by Kubrick *OR* Polanski?
- ```
SELECT Actor
FROM Movies
WHERE Director='Kubrick' OR Director='Polanski'
```
- 
- Can this be defined by a *single* rule?
- NO!

## Disjunction in queries cont'd

- Solution: Disjunction can be represented by more than one rule.
- $\text{answer}(\text{act}) \text{ :- movies}(\text{tl}, \text{dir}, \text{act}), \text{dir} = \text{'Kubrick'}$
- $\text{answer}(\text{act}) \text{ :- movies}(\text{tl}, \text{dir}, \text{act}), \text{dir} = \text{'Polanski'}$
- Semantics: compute answers to each of the rules, and then take their *union*.
- SQL has another syntax for that:

```
SELECT Actor
FROM Movies
WHERE Director='Kubrick'
UNION
SELECT Actor
FROM Movies
WHERE Director='Polanski'
```

## Disjunction in queries cont'd

- How to translate a query with disjunction into relational algebra?
- $\text{answer}(\text{act}) \text{ :- movies}(\text{tl}, \text{dir}, \text{act}), \text{ dir} = \text{'Kubrick'}$   
is translated into  $Q_1 = \pi_{\text{actor}}(\sigma_{\text{director}=\text{Kubrick}}(\text{Movies}))$
- $\text{answer}(\text{act}) \text{ :- movies}(\text{tl}, \text{dir}, \text{act}), \text{ dir} = \text{'Polanski'}$   
is translated into  $Q_2 = \pi_{\text{actor}}(\sigma_{\text{director}=\text{Polanski}}(\text{Movies}))$
- The whole query is translated into  $Q_1 \cup Q_2$

$$\pi_{\text{actor}}(\sigma_{\text{director}=\text{Kubrick}}(\text{Movies})) \cup \pi_{\text{actor}}(\sigma_{\text{director}=\text{Polanski}}(\text{Movies}))$$

## Union in relational algebra

- Another operation of relational algebra: union
- $R \cup S$  is the union of relations  $R$  and  $S$
- $R$  and  $S$  must have the same set of attributes.
- We now have four relational algebra operations:

$$\pi, \sigma, \times, \cup$$

(and of course  $\bowtie$  which is definable from  $\pi, \sigma, \times$ )

- This fragment is called *positive relational algebra*, or SPJU-queries (select-project-join-union)

## Interaction of relational algebra operators

- $\pi_{\vec{A}}(R \cup S) = \pi_{\vec{A}}(R) \cup \pi_{\vec{A}}(S)$
- $\sigma_c(R \cup S) = \sigma_c(R) \cup \sigma_c(S)$
- $(R \cup S) \times T = R \times T \cup S \times T$
- $T \times (R \cup S) = T \times R \cup T \times S$

## SPJU queries

*Every SPJU query is equivalent to a union of SPJ queries.*

Because: one propagates the union operation.

Example:

$$\begin{aligned} & \pi_A(\sigma_c((R \times (S \cup T)) \cup W)) \\ = & \pi_A(\sigma_c((R \times S) \cup (R \times T) \cup W)) \\ = & \pi_A(\sigma_c(R \times S) \cup \sigma_c(R \times T) \cup \sigma_c(W)) \\ = & \pi_A(\sigma_c(R \times S)) \cup \pi_A(\sigma_c(R \times T)) \cup \pi_A(\sigma_c(W)) \end{aligned}$$

## Equivalences

- Positive relational algebra (SPJU queries)
- = Unions of SPJ queries
- = queries defined by multiple rules
- = SQL SELECT-FROM-WHERE-UNION
- = unions of conjunctive queries
- = queries defined with  $\exists, \wedge, \vee$

Question: is INTERSECTION an SPJU query?

That is, given  $R, S$  with the same set of attributes, find  $R \cap S$ .

## More on union

- Relation  $R_1$ : father, child
 

| $R_1$ | father  | child     |
|-------|---------|-----------|
|       | George  | Elizabeth |
|       | Philip  | Charles   |
|       | Charles | William   |
- Relation  $R_2$ : mother, child
 

| $R_2$ | mother    | child   |
|-------|-----------|---------|
|       | Elizabeth | Charles |
|       | Elizabeth | Andrew  |
- We want their union, which should be the “parent-child” relation.
- But we cannot use  $R_1 \cup R_2$  because  $R_1$  and  $R_2$  have different attributes!
- Hence we must *rename* attributes.

## Renaming

- Let  $R$  be a relation that has attribute  $A$  but does *not* have attribute  $B$ .
- $\rho_{B \leftarrow A}(R)$  is the same relation as  $R$  except that  $A$  is renamed to be  $B$ .
- 

$$\rho_{\text{parent} \leftarrow \text{father}} \left( \begin{array}{c} \frac{\text{father} \quad \text{child}}{\text{George} \quad \text{Elizabeth}} \\ \text{Philip} \quad \text{Charles} \\ \text{Charles} \quad \text{William} \end{array} \right) = \left( \begin{array}{c} \frac{\text{parent} \quad \text{child}}{\text{George} \quad \text{Elizabeth}} \\ \text{Philip} \quad \text{Charles} \\ \text{Charles} \quad \text{William} \end{array} \right)$$

$$\rho_{\text{parent} \leftarrow \text{mother}} \left( \begin{array}{c} \frac{\text{mother} \quad \text{child}}{\text{Elizabeth} \quad \text{Charles}} \\ \text{Elizabeth} \quad \text{Andrew} \end{array} \right) = \left( \begin{array}{c} \frac{\text{parent} \quad \text{child}}{\text{Elizabeth} \quad \text{Charles}} \\ \text{Elizabeth} \quad \text{Andrew} \end{array} \right)$$

## Renaming

The desired union now is:

$$\rho_{\text{parent} \leftarrow \text{father}}(R_1) \cup \rho_{\text{parent} \leftarrow \text{mother}}(R_2)$$

and it produces:

| parent    | child     |
|-----------|-----------|
| George    | Elizabeth |
| Philip    | Charles   |
| Charles   | William   |
| Elizabeth | Charles   |
| Elizabeth | Andrew    |

## Renaming in SQL

- New attribute names can be introduced in SELECT using keyword AS.
- SELECT Father AS Parent, Child  
FROM R1

```
SELECT Mother AS Parent, Child
FROM R2
```

- The union of these queries can be taken, as they have the same set of attributes:

```
SELECT Father AS Parent, Child
FROM R1
UNION
SELECT Mother AS Parent, Child
FROM R2
```

## Queries with “All”

- Find directors whose movies are playing in all theaters.

$$\{ \text{dir} \mid \forall (\text{th}, \text{tl}') \in \text{Schedule} \exists \text{tl}, \text{act} \text{ Schedule}(\text{th}, \text{tl}) \wedge \text{Movies}(\text{tl}, \text{dir}, \text{act}) \}$$

- What does it actually mean?
- To understand this, we revisit rule-based queries, and write them in logical notation.

## Rules revisited

- By now, this query is very familiar:
- $\text{answer}(\text{th}) \text{ :- movies}(\text{tl}, \text{'Polanski'}, \text{act}), \text{schedule}(\text{th}, \text{tl})$
- What does it actually mean?
- It asks, for each theater (th): “Does there exist a movie (tl) and an actor (act) such that (th,tl) is in Schedule and (tl, 'Polanski', act) is in Movies?”
- This can be stated using notation from mathematical logic:

$$Q(\text{th}) = \exists \text{tl} \exists \text{act} \text{Movies}(\text{tl}, \text{'Polanski'}, \text{act}) \wedge \text{Schedule}(\text{th}, \text{tl})$$

## Other queries in logical notation

- $\text{answer}(\text{th}) \text{ :- movies}(\text{tl}, \text{dir}, \text{'Nicholson'}), \text{schedule}(\text{th}, \text{tl})$

- 

$$Q(\text{th}) = \exists \text{tl} \exists \text{dir} \text{Movies}(\text{tl}, \text{dir}, \text{'Nicholson'}) \wedge \text{Schedule}(\text{th}, \text{tl})$$

- In general, every single-rule query can be written in the logical notation using only:

existential quantification  $\exists$ , and

logical conjunction  $\wedge$  (AND)

## SPJU queries in logical form

- Find actors who played in movies directed by Kubrick *OR* Polanski.

- Rule-based query:

answer(act) :- movies(tl,dir,act), dir='Kubrick'

answer(act) :- movies(tl,dir,act), dir='Polanski'

- Logical notation:

$$Q(\text{act}) = \exists \text{tl} \exists \text{dir} \left( \begin{array}{l} \text{Movies}(\text{tl},\text{dir},\text{act}) \\ \wedge (\text{dir}=\text{'Kubrick'} \vee \text{dir}=\text{'Polanski'}) \end{array} \right)$$

- New element here: logical disjunction  $\vee$  (OR)
- SPJU queries can be written in logical notation using:

existential quantifiers  $\exists$

conjunction  $\wedge$  and disjunction  $\vee$

## Queries with “for all”

- $\{ \text{dir} \mid \forall (\text{th}, \text{tl}') \in \text{Schedule} \exists \text{tl}, \text{act} \text{ Schedule}(\text{th}, \text{tl}) \wedge \text{Movies}(\text{tl}, \text{dir}, \text{act}) \}$
- New element here: universal quantification “for all”  $\forall$
- $\forall x F(x) = \neg \exists x \neg F(x)$
- So really the new element is: *negation*
- One has to be careful with negation: what is the meaning of

$$\{x \mid \neg R(x)\}$$

- It seems to say: give us everything that is *not* in the database. But this is an *infinite* set!

## Queries with “all” and negation cont’d

- Safety: a query written in logical notation is *safe* if it is guaranteed to return finite results on all databases.
- Clearly this has to be enforced in practical languages.
- Bad news: No algorithm can possibly check if a query is safe.
- A bit of good news: All SPJ and SPJU queries are safe.  
Because: everything that occurs in the output must have occurred in the input: no new elements are created.
- So we have to figure out how to handle negation.

## Relational Calculus

- Relational calculus: queries written in the logical notation using:
  - relation names (e.g., Movies)
  - constants (e.g., 'Nicholson')
  - conjunction  $\wedge$ , disjunction  $\vee$
  - negation  $\neg$
  - existential quantifiers  $\exists$
  - universal quantifiers  $\forall$
- $\wedge, \exists, \neg$  suffice:
$$\forall x F(x) = \neg \exists x \neg F(x)$$
$$F \vee G = \neg(\neg F \wedge \neg G)$$
- Another name for it: first-order predicate logic.

## Relational Calculus cont'd

- Bound variable: a variable  $x$  that occurs in  $\exists x$  or  $\forall x$
- Free variable: a variable that is not bound.
- Free variables are those that go into the output of a query.
- Two ways to write a query:

$Q(\vec{x}) = F$ , where  $\vec{x}$  is the tuple of free variables

$\{\vec{x} \mid F\}$

- Examples:

$\{x, y \mid \exists z (R(x, z) \wedge S(z, y))\}$

$\{x \mid \forall y R(x, y)\}$

- Queries without free variables are called *Boolean queries*.
- Their output is *true* or *false*

$\forall x R(x, x)$

$\forall x \exists y R(x, y)$

## Safe Relational Calculus

- A relational calculus query  $Q(\vec{x})$  is safe if it always returns a finite result.
- Examples of safe queries:
  - Any Boolean query
  - Any SPJ or SPJU query
- Examples of unsafe queries:
  - $\{x \mid \neg R(x)\}$
  - $\{x, y \mid \text{Movies}(x, \text{Polanski}, \text{Nicholson}) \vee \text{Movies}(\text{Chinatown}, \text{Polanski}, y)\}$
- Safe relational calculus = set of safe relational calculus queries.
- But safety cannot be checked algorithmically!
- Still, we can describe this language.

## Difference

- If  $R$  and  $S$  are two relations with the same set of attributes, then  $R - S$  is their difference:

The set of all tuples that occur in  $R$  but not in  $S$ .

- Example:

$$\begin{array}{cc} \underline{A \quad B} \\ a1 \quad b1 \\ a2 \quad b2 \\ a3 \quad b3 \end{array} - \begin{array}{cc} \underline{A \quad B} \\ a2 \quad b2 \\ a3 \quad b3 \\ a4 \quad b4 \end{array} = \begin{array}{cc} \underline{A \quad B} \\ a1 \quad b1 \end{array}$$

# Relational Algebra

- Includes operations  $\pi, \sigma, \times, \cup, -, \rho$

Fundamental Theorem of Relational Database Theory

Safe Relational Calculus = Relational Algebra

- We won't give a formal proof of this statement, but try to explain why it is true.  
You'll also see some examples of relational algebra programming along the way.

## From Relational algebra to Safe relational calculus

- Show that relational algebra can be expressed by relational calculus (and it is certainly safe)
- Each expression  $e$  producing an  $n$ -attribute relation is translated into a formula  $F_e(x_1, \dots, x_n)$
- $R \rightarrow R(x_1, \dots, x_n)$
- $\sigma_c(R) \rightarrow R(x_1, \dots, x_n) \wedge c$

For example, if  $R$  has attributes  $A, B$  then  $\sigma_{A=B}(R)$  is translated into  $(R(x_1, x_2) \wedge x_1 = x_2)$ .

## From Relational algebra to Safe relational calculus cont'd

- If  $R$  has attributes  $A_1, \dots, A_n, B_1, \dots, B_m$ , then

$$\pi_{A_1, \dots, A_n}(R)$$

is translated into

$$\exists y_1, \dots, y_m R(x_1, \dots, x_n, y_1, \dots, y_m)$$

Important: it is the attributes that are *not* projected that are quantified.

Example: for  $R$  with attributes  $A, B$ ,  $\pi_A(R)$  is  $\exists x_2 R(x_1, x_2)$ .

- $R \times S$  is translated into  $R(x_1, \dots, x_n) \wedge S(y_1, \dots, y_m)$   
(note that all the variables are distinct; hence the output will have  $n + m$  attributes)

## From Relational algebra to Safe relational calculus cont'd

- If  $R$  and  $S$  both have the same attributes, then  $R \cup S$  is translated into  $R(x_1, \dots, x_n) \vee S(x_1, \dots, x_n)$   
(note that all the variables are the same, hence the output will have  $n$  attributes)
- If  $R$  and  $S$  both have the same attributes, then  $R - S$  is translated into  $R(x_1, \dots, x_n) \wedge \neg S(x_1, \dots, x_n)$   
(note that all the variables are the same, hence the output again will have  $n$  attributes)

## Getting ready for the calculus to algebra translation

- Active domain of a relation: the set of all constants that occur in it.

- Example: active domain of 

|       |       |       |
|-------|-------|-------|
| $R_1$ | $A$   | $B$   |
|       | $a_1$ | $b_1$ |
|       | $a_2$ | $b_2$ |

 is  $\{a_1, a_2, b_1, b_2\}$ .

- Computing the active domain of  $R$ .  
Suppose  $R$  has attributes  $A_1, \dots, A_n$ .

$$\text{ADOM}(R) = \rho_{B \leftarrow A_1}(\pi_{A_1}(R)) \cup \dots \cup \rho_{B \leftarrow A_n}(\pi_{A_n}(R))$$

- It is a relation with one attribute  $B$ .
- Similarly we can compute

$$\text{ADOM}(R_1, \dots, R_k) = \text{ADOM}(R_1) \cup \dots \cup \text{ADOM}(R_k)$$

## From safe relational calculus to relational algebra

- A safe query over relations  $R_1, \dots, R_n$  cannot produce an element outside of  $\text{ADOM}(R_1, \dots, R_n)$
- That is, for a safe query  $Q$ ,

$$\text{ADOM}(Q(R_1, \dots, R_n)) \subseteq \text{ADOM}(R_1, \dots, R_n)$$

- Because: every element outside of  $\text{ADOM}(R_1, \dots, R_n)$  “looks” like any other element: so if one is in the output, then all are, and hence the query isn’t safe.
- We thus translate relational calculus queries evaluated within  $\text{ADOM}(R_1, \dots, R_n)$  into relational algebra queries.
- Each relational calculus formula  $F(x_1, \dots, x_n)$  is translated into an expression  $E_F$  that produces a relation with  $n$  attributes.

## From safe relational calculus to relational algebra: translation

- Easy cases (for  $R$  with attributes  $A_1, \dots, A_n$ ):

$$R(x_1, \dots, x_n) \rightarrow R$$

$$\exists x_1 R(x_1, \dots, x_n) \rightarrow \pi_{A_2, \dots, A_n}(R)$$

- Not so easy cases:

- condition  $c(x_1, \dots, x_n)$  is translated into

$$\sigma_c(\text{ADOM} \times \dots \times \text{ADOM})$$

E.g.,  $x_1 = x_2$  is translated into  $\sigma_{x_1=x_2}(\text{ADOM} \times \text{ADOM})$

- Negation  $\neg R(\vec{x}) \rightarrow \text{ADOM} \times \dots \times \text{ADOM} - R$

That is, we only compute the tuples of elements from the database that do not belong to  $R$

## From safe relational calculus to relational algebra cont'd

- The hardest case: disjunction
- Let both  $R$  and  $S$  have two attributes.
- Relational calculus query:

$$Q(x, y, z) = R(x, y) \vee S(x, z)$$

- Its result has three attributes, and consists of tuples  $(x, y, z)$  such that either:

$$(x, y) \in R, z \in \text{ADOM}, \text{ or}$$

$$(x, z) \in S, y \in \text{ADOM}$$

- The first one is simply  $R \times \text{ADOM}$
- The second one is more complex:  $\pi_{\#1, \#3, \#5}(\sigma_{\#1=\#4 \wedge \#2=\#5}(S \times \text{ADOM} \times S))$
- Thus,  $Q$  is translated into

$$R \times \text{ADOM} \cup \pi_{\#1, \#3, \#5}(\sigma_{\#1=\#4 \wedge \#2=\#5}(S \times \text{ADOM} \times S))$$

## Queries with “all” in relational algebra revisited

- Find directors whose movies are playing in all theaters.

$$\{ \text{dir} \mid \forall (\text{th}, \text{tl}') \in \text{Schedule} \exists \text{tl}, \text{act} \text{ Schedule}(\text{th}, \text{tl}) \wedge \text{Movies}(\text{tl}, \text{dir}, \text{act}) \}$$

- Define:

$$T_1 = \pi_{\text{theater}}(S) \quad T_2 = \pi_{\text{theater}, \text{director}}(M \bowtie S)$$

(to save space, we use  $M$  for Movies and  $S$  for Schedule)

- $T_1$  has all theaters,  $T_2$  has all directors and theaters where their movies are playing.
- Our query is:

$$\{d \mid \forall t \in T_1 \quad (t, d) \in T_2\}$$

## Queries with “all” cont’d

$$\{d \mid \forall t \in T_1 \quad T_2(t, d)\}$$

is rewritten to

$$\{d \mid \neg(\exists t \in T_1 (t, d) \notin T_2)\}$$

Hence, the answer to the query is

$$\pi_{\text{director}}(M) - V$$

where  $V = \{d \mid (\exists t \in T_1 (t, d) \notin T_2)\}$ .

Pairs (theater, director) not in  $T_2$  are

$$T_1 \times \pi_{\text{director}}(M) - T_2$$

Thus

$$V = \pi_{\text{director}}(T_1 \times \pi_{\text{director}}(M) - T_2)$$

## Queries with “all” cont’d

- Reminder: the query is  
Find directors whose movies are playing in all theaters.
- Putting everything together, the answer is:

$$\pi_{\text{director}}(M) - \pi_{\text{director}}(\pi_{\text{theater}}(S) \times \pi_{\text{director}}(M) - \pi_{\text{theater,director}}(M \bowtie S))$$

- This is much less intuitive than the logical description of the query.
- Indeed, procedural languages are not nearly as comprehensible as declarative.

## For all and negation in SQL

- Two main mechanisms: subqueries, and Boolean expressions
- Subqueries are often more natural

- SQL syntax for  $R \cap S$ :

```
R INTERSECT S
```

- SQL syntax for  $R - S$ :

```
R EXCEPT S
```

- Find all actors who are  
not directors:

```
SELECT Actor AS Person
FROM Movies
EXCEPT
SELECT Director AS Person
FROM Movies
```

also directors:

```
SELECT Actor AS Person
FROM Movies
INTERSECT
SELECT Director AS Person
FROM Movies
```

## For all and negation in SQL cont'd

- Find directors whose movies are playing in all theaters.
- SQL's way of saying this: Find directors such that there does not exist a theater where their movies do not play.

```
SELECT M1.Director
FROM Movies M1
WHERE NOT EXISTS (SELECT S.Theater
 FROM Schedule S
 WHERE NOT EXISTS (SELECT M2.Director
 FROM Movies M2
 WHERE M2.Title=S.Title
 AND
 M1.Director=M2.Director))
```

## For all and negation in SQL cont'd

Same query using EXCEPT.

```
SELECT M.Director
FROM Movies M
WHERE NOT EXISTS (SELECT S.Theater
 FROM Schedule S
 EXCEPT
 SELECT S1.Theater
 FROM Schedule S1, Movies M1
 WHERE S1.Title=M1.Title
 AND M1.Director=M.Director)
```

- Other conditions: IN, NOT IN, EXISTS

More examples of nested queries: using EXISTS and IN

Find directors whose movies are playing at Le Champo:

```
SELECT M.Director
FROM Movies M
WHERE EXISTS (SELECT *
 FROM Schedule S
 WHERE S.Title=M.Title
 AND S.Theater='Le Champo')
```

```
SELECT M.Director
FROM Movies M
WHERE M.Title IN (SELECT S.Title
 FROM Schedule S
 WHERE S.Theater='Le Champo')
```

More examples of nested queries: using NOT IN

Find actors who did not play in a movie by Kubrick.

```
SELECT M.Actor
FROM Movies M
WHERE M.Actor NOT IN
 (SELECT M1.Actor
 FROM Movies M1
 WHERE M1.Director='Kubrick')
```

The subquery finds actors playing in some movie by Kubrick; the top two lines take the complement of that.

## Database Constraints

- So far we assumed that the *title* attribute identifies a movie.
- But this may not be the case:

| title   | director | actor    | length |
|---------|----------|----------|--------|
| Dracula | Browning | Lugosi   | 75     |
| Dracula | Fischer  | Lee      | 83     |
| Dracula | Badham   | Langella | 109    |
| Dracula | Coppola  | Oldman   | 130    |

- Database constraints: provide additional semantic information about the data.
- Most common ones: functional and inclusion dependencies, and their special cases: **keys** and **foreign keys**.

Why didn't we have this problem before?

- What does it mean that *title* identifies a movie uniquely? A movie may have several actors, but at most one director – hence it means that *title* determines the value of the *director* attribute.
- This is expressed as a *functional dependency*

$\text{title} \rightarrow \text{director}$

- In general, a relation  $R$  satisfies a functional dependency  $A \rightarrow B$ , where  $A$  and  $B$  are attributes, if for every two tuples  $t_1, t_2$  in  $R$ :

$$\pi_A(t_1) = \pi_A(t_2) \quad \text{implies} \quad \pi_B(t_1) = \pi_B(t_2)$$

## Functional dependencies and keys

- More generally, a functional dependency is  $X \rightarrow Y$  where  $X, Y$  are sequences of attributes. It holds in a relation  $R$  if for every two tuples  $t_1, t_2$  in  $R$ :

$$\pi_X(t_1) = \pi_X(t_2) \quad \text{implies} \quad \pi_Y(t_1) = \pi_Y(t_2)$$

- A very important special case: *keys*
- Let  $K$  be a set of attributes of  $R$ , and  $U$  the set of *all* attributes of  $R$ . Then  $K$  is a key if  $R$  satisfies functional dependency  $K \rightarrow U$ .
- In other words, a set of attributes  $K$  is a key in  $R$  if for any two tuples  $t_1, t_2$  in  $R$ ,

$$\pi_K(t_1) = \pi_K(t_2) \quad \text{implies} \quad t_1 = t_2$$

- That is, a key is a set of attributes that uniquely identify a tuple in a relation.

## keys cont'd

- Consider

| title   | director | actor    | length | year |
|---------|----------|----------|--------|------|
| Dracula | Browning | Lugosi   | 75     | 1931 |
| Dracula | Fischer  | Lee      | 83     | 1958 |
| Dracula | Badham   | Langella | 109    | 1979 |
| Dracula | Coppola  | Oldman   | 130    | 1992 |

- Then the following are keys:

(Title, Year)

(Title, Actor)

(Title, Length)

## Inclusion constraints

- We expect every Title listed in Schedule to be present in Movies.
- These are *referential* integrity constraints: they talk about attributes of one relation (Schedule) but refer to values in another one (Movies).
- These particular constraints are called *inclusion dependencies* (ID).
- Formally, we have an inclusion dependency  $R[A] \subseteq S[B]$  when every value of attribute  $A$  in  $R$  also occurs as a value of attribute  $B$  in  $S$ :

$$\pi_A(R) \subseteq \pi_B(S)$$

- As with keys, this extends to sets of attributes, but they must have the same number of attributes.
- There is an inclusion dependency  $R[A_1, \dots, A_n] \subseteq S[B_1, \dots, B_n]$  when

$$\pi_{A_1, \dots, A_n}(R) \subseteq \pi_{B_1, \dots, B_n}(S)$$

## Foreign keys

- Most often inclusion constraints occur as a part of a *foreign key*
- Foreign key is a conjunction of a key and an ID:

$$R[A_1, \dots, A_n] \subseteq S[B_1, \dots, B_n] \quad \text{and} \\ \{B_1, \dots, B_n\} \rightarrow \text{all attributes of } S$$

- Meaning: we find a key for relation  $S$  in relation  $R$ .
- Example: Suppose we have relations:  
Employee(EmplId, Name, Dept, Salary)  
ReportsTo(Empl1, Empl2).  
  
• We expect both Empl1 and Empl2 to be found in Employee; hence:  
ReportsTo[Empl1]  $\subseteq$  Employee[EmplId]  
ReportsTo[Empl2]  $\subseteq$  Employee[EmplId].
- If EmplId is a key for Employee, then these are foreign keys.