“Those who cannot remember the past are doomed to repeat it”
THIS LECTURE

• Different Data Models (particular IMS, Codasyl, Relations)
• Brief Relational Algebra Intro
DATABASE ABSTRACTION LAYERS

Data Independence

Logical Data Independence

Physical Data Independence

Changes at one layer do not affect another layer!
MODIFIED ZOO DATA MODEL

animals(name, species, age, feed time)

cages(no, size, bldg)

keepers(name, address)

Slightly different than last time:

- Each animal in 1 cage, multiple animals share a cage
- Each animal cared for by 1 keeper, keepers care for multiple animals
- Animals have feed times, not cages
EXAMPLE IMS HIERARCHY

Keepers segment

<table>
<thead>
<tr>
<th>A1 Segment</th>
<th>A2 Segment</th>
<th>A3 Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 Segment</td>
<td>C2 Segment</td>
<td>C3 Segment</td>
</tr>
</tbody>
</table>

Jane (keeper) (1)
Sam, salamander, … (2)
1, 100sq ft, … (3)
Mike, giraffe, … (4)
2, 1000sq ft, … (5)
Sally, student, … (6)
1, 100sq ft, … (7)
Joe (keeper) (8)

- DB administrator has to choose a physical representation for each segment
- Can create an index on the root segment, other segments can only be iterated through.
- Root segments can be sequential, or hashed, or tree-based indexes.
EXAMPLE ALTERNATIVE IMS HIERARCHY

Keepers segment

<table>
<thead>
<tr>
<th>C1 Segment</th>
<th>C2 Segment</th>
<th>C3 Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 Segment</td>
<td>A2 Segment</td>
<td>A3 Segment</td>
</tr>
</tbody>
</table>

Jane (keeper) (1)
1, 100sq ft, … (2)
   Sam, salamander, … (3)
2, 1000sq ft, … (4)
   Mike, giraffe, … (5)
   Albert, giraffe,…(6)

Joe (keeper) (7)
1, 100sq ft, … (8)
IMS COMMANDS

GU (segment type, predicate) - *get unique*

- Optional predicate can be used when root is indexed
- Finds the first segment satisfying a predicate, and set cursor there

GN (seg, pred) - *get next key*

- in hierarchical order
- moving on to other parents
- optional predicate
- search begins from last GU/GN call

GNP (seg) - *get next record of the specified segment type, stopping when leaving current parent*

D - delete this record

I - add a new record
EXAMPLE IMS PROGRAMS

Find the cages that Jane keeps
Hierarchy: Keepers → Cage → Animals

GetUnique(Keepers, name = "Jane")
Until done:
    cageid = GetNextParent (cages).no
    print cageid
Find the keepers that take care of cage 6:
Hierarchy: Keepers -> Cage -> Animals

- **GU** (segment type, predicate) - *get unique*
  - Optional predicate can be used when root is indexed
  - Finds the first segment satisfying a predicate, and sets cursor there
- **GN** (seg, pred) - *get next key*
  - In hierarchical order
  - Moving on to other parents
  - Optional predicate
  - Search begins from last GU/GN call
- **GNP** (seg) - *get next record of the specified segment type, stopping when leaving current parent*
EXAMPLE IMS PROGRAMS

Find the keepers that take care of cage 6:

Solution A

cage = GNP(cages, id = 6)
KeeperName = GN(keepers)
print KeeperName

Solution B

Until done:

KeeperName = GN(keepers)
cage = GNP(cages, id = 6)
if(cage != null)
    print KeeperName

Solution C

KeeperName = GU(keepers).name
cage = GNP(cages, id = 6)
if(cage != null)
    print KeeperName

http://clicker.csail.mit.edu/6.814/

• GU (segment type, predicate) - get unique
  – Optional predicate can be used when root is indexed
  – Finds the first segment satisfying a predicate, and set
cursor there

• GN (seg, pred) - get next key
  – in hierarchical order
  – moving on to other parents
  – optional predicate
  – search begins from last GU/GN call

• GNP (seg) - get next record of the specified
  segment type, stopping when leaving current parent
IMS PROBLEMS

1. Duplication of data

2. Painful low level programming interface – have to program the search algorithm

3. Limited physical data independence
   change root from indexed to hash --- programs that do GN on the root segment will fail cannot do inserts into sequential root structure  → One of Codd’s key points!!! This is terrible system design

4. Limited logical data independence -- schemas change, would like it if programs didn't have to
CODASYL
EXAMPLE CODASYL HIERARCHY

- *livesin* and *caredforby* are called “sets” (i.e., are actually not sets but 1-n relationships)
- Need one entry point (a record that is not the child in any set)
- Only binary relationships between entities
- Records can either be hashes (allowing equality lookup) or sorted (“clustered”) according to some key (allowing a range lookup).
- Follow pointers to find records the interest -- "record at a time" programming like in IMS
EXAMPLE CODASYL PROGRAM

Find the cages that Joe keeps

Find keepers (name = 'Joe')
Until done:
  Find next animal in caredforby
  Find cage in livesin
CODASYL PROBLEMS

• **Incredibly complex** — “Navigational Programming”

• Programs are very tied to the representation above, which means there is no physical or logical data independence
  • (Can't change schema w/out changing programs; can't change physical representation either b/c different index types might or might not support different operations)

• Implementations had bad properties -- like you had to load all of the data at once.

Some of this could have been fixed by adding a high level language to CODASYL (this was proposed)
RELATIONAL MODEL
**RELATIONAL MODEL - TERMS**

Account =

<table>
<thead>
<tr>
<th>Table name</th>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>Brighton</td>
<td>A-201</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>500</td>
<td></td>
</tr>
</tbody>
</table>

Terms

- Tables → Relations
- Columns → Attributes
- Rows → Tuples
- Schema (e.g.: Acct_Schema = (bname, acct_no, balance))
WHY ARE THEY CALLED RELATIONS?

Relation:

- \( R \subseteq D_1 \times \ldots \times D_n \)
- \( D_1, D_2, \ldots, D_n \) are domains

Example: AddressBook \( \subseteq \) string \( \times \) string \( \times \) integer
WHY ARE THEY CALLED RELATIONS?

**Relation:**

- $R \subseteq D_1 \times \ldots \times D_n$
- $D_1, D_2, \ldots, D_n$ are domains

*Example:* AddressBook $\subseteq$ string $\times$ string $\times$ integer

**Tuple:** $t \in R$

*Example:* $t = ("Mickey Mouse", "Main Street", 4711)$
WHY ARE THEY CALLED RELATIONS?

Relation:

- $R \subseteq D_1 \times \ldots \times D_n$
- $D_1, D_2, \ldots, D_n$ are domains

Example: AddressBook $\subseteq$ string x string x integer

Tuple: $t \in R$
Example: $t = ("Mickey Mouse", "Main Street", 4711)$

Schema: associates labels to domains
Example:

$AddrBook: \{[Name: string, Address: string, Tel#:integer]\}$
<table>
<thead>
<tr>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>500</td>
</tr>
</tbody>
</table>

Considered equivalent to...

\[
\{ (\text{Downtown}, A-101, 500), \\
(\text{Brighton}, A-201, 900), \\
(\text{Brighton}, A-217, 500) \}
\]

Relational database semantics are defined in terms of mathematical relations (i.e., sets)
# KEYS AND RELATIONS

## Kinds of keys

- **Superkeys:**
  set of attributes of table for which every row has distinct set of values
- **Candidate keys:**
  “minimal” superkeys
- **Primary keys:**
  DBA-chosen candidate key (marked in schema by underlining)

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Author</th>
<th>Edition</th>
<th>Publisher</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>0439708184</td>
<td>Harry Potter</td>
<td>J.K. Rowling</td>
<td>1</td>
<td>Scholastic</td>
<td>$6.70</td>
</tr>
<tr>
<td>0545663261</td>
<td>Mockingjay</td>
<td><a href="https://en.wikipedia.org/wiki/Suzanne_Collins">Suzanne Collins</a></td>
<td>1</td>
<td>Scholastic</td>
<td>$7.39</td>
</tr>
</tbody>
</table>
KEYS AND RELATIONS

Kinds of keys

- **Superkeys**: set of attributes of table for which every row has distinct set of values
- **Candidate keys**: “minimal” superkeys
- **Primary keys**: DBA-chosen candidate key (marked in schema by underlining)

**Act as Integrity Constraints**

i.e., guard against illegal/invalid instance of given schema

e.g., \( \text{Branch} = (\text{bname}, \text{bcity}, \text{assets}) \) \( \notin \)

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>Brooklyn</td>
<td>5M</td>
</tr>
<tr>
<td>Brighton</td>
<td>Boston</td>
<td>3M</td>
</tr>
</tbody>
</table>

Invalid!!
RELATIONAL ALGEBRA
FORMAL DEFINITION OF REL. ALGEBRA

- **Selection**: $\sigma_{\text{pred}}(R1)$
- **Projection**: $\Pi_A(R1)$
- **Cartesian Product**: $R1 \times R2$
- **Rename**: $\rho_V(R1), \rho_A \leftarrow B(R1)$
- **Union**: $R1 \cup R2$
- **Minus**: $R1 - R2$

---

- **Join**: $\bowtie = \sigma_{R1\.id = R2\.id}(R1 \times R2)$
CLOSURE PROPERTY / COMPOSABILITY
Professor(Person-ID:integer, Name:varchar(30), Level:varchar(2))
Student(Student-ID:integer, Name:varchar(30), Semester:integer)
Lecture(Course-ID:varchar(10), Title:varchar(50), CP:float)
Gives(Person-ID:integer, Course-ID:varchar(10))
Attends(Student-ID:integer, Course-ID:varchar(10))
Tests(Student-ID:integer, Course-ID:varchar(10), Person-ID:integer, Grade:char(2))
### SELECTION AND PROJECTION

Professor\((\text{Person-ID}: \text{integer}, \text{Name}: \text{varchar}(30), \text{Level}: \text{varchar}(2))\)

Student\((\text{Student-ID}: \text{integer}, \text{Name}: \text{varchar}(30), \text{Semester}: \text{integer})\)

#### Selection

\[ \sigma_{\text{Semester} > 10} (\text{Student}) \]

<table>
<thead>
<tr>
<th>Student-ID</th>
<th>Name</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>24002</td>
<td>Xenokrates</td>
<td>18</td>
</tr>
<tr>
<td>25403</td>
<td>Jonas</td>
<td>12</td>
</tr>
</tbody>
</table>

#### Projection

\[ \Pi_{\text{Level}} (\text{Professor}) \]

<table>
<thead>
<tr>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
</tr>
<tr>
<td>AP</td>
</tr>
</tbody>
</table>
CARTESIAN PRODUCT

\[ L \times R = \begin{array}{cccccc}
A & B & C & D & E \\
\hline
a_1 & b_1 & c_1 & d_1 & e_1 \\
a_2 & b_2 & c_2 & d_2 & e_2 \\
\end{array} \]
CARTESIAN PRODUCT (CTD.)

Professor x attends

<table>
<thead>
<tr>
<th>Person-ID</th>
<th>Name</th>
<th>Level</th>
<th>Room</th>
<th>Student-ID</th>
<th>Course-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2125</td>
<td>Ugur</td>
<td>FP</td>
<td>226</td>
<td>26120</td>
<td>5001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2125</td>
<td>Ugur</td>
<td>FP</td>
<td>226</td>
<td>29555</td>
<td>5001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2137</td>
<td>Jeff</td>
<td>AP</td>
<td>7</td>
<td>29555</td>
<td>5001</td>
</tr>
</tbody>
</table>

- Huge result set \((n \times m)\)
- Usually only useful in combination with a selection (-> Join)
Natural Join

Two relations:

- \( R(A_1, \ldots, A_m, B_1, \ldots, B_k) \)
- \( S(B_1, \ldots, B_k, C_1, \ldots, C_n) \)

\[
R \bowtie S = \prod_{A_1, \ldots, A_m, R.B_1, \ldots, R.B_k, C_1, \ldots, C_n} (\sigma_{R.B_1 = S.B_1 \land \ldots \land R.B_k = S.B_k}(R \times S))
\]
THREE-WAY NATURAL JOIN

(Student $\bowtie$ attends) $\bowtie$ Lecture

<table>
<thead>
<tr>
<th>Student-ID</th>
<th>Name</th>
<th>Semester</th>
<th>Course-Nb</th>
<th>Title</th>
<th>CP</th>
<th>Person-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>26120</td>
<td>Mike</td>
<td>10</td>
<td>6.814</td>
<td>Database System</td>
<td>2</td>
<td>9999</td>
</tr>
<tr>
<td>27550</td>
<td>Dave</td>
<td>12</td>
<td>6.006</td>
<td>Algorithms</td>
<td>2</td>
<td>2134</td>
</tr>
<tr>
<td>28106</td>
<td>Alex</td>
<td>3</td>
<td>6.141</td>
<td>Robotics</td>
<td>3</td>
<td>2126</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

...
THETA-JOIN

Two Relations:

- \( R(A_1, ..., A_n) \)
- \( S(B_1, ..., B_m) \)

\[
R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)
\]
JOIN VARIANTS

• natural join

\[
\begin{array}{ccc}
L & R & \rightarrow \\
A & B & C \\
a_1 & b_1 & c_1 \\
a_2 & b_2 & c_2 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{Result} \\
A & B & C & D & E \\
a_1 & b_1 & c_1 & d_1 & e_1 \\
c_3 & d_2 & e_2 \\
\end{array}
\]

• left outer join

\[
\begin{array}{ccc}
L & R & \leftarrow \\
A & B & C \\
a_1 & b_1 & c_1 \\
a_2 & b_2 & c_2 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{Result} \\
A & B & C & D & E \\
a_1 & b_1 & c_1 & d_1 & e_1 \\
a_2 & b_2 & c_2 & - & - \\
\end{array}
\]
JOIN VARIANTS

- right outer join

<table>
<thead>
<tr>
<th>L</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td></td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
<td>c₂</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>c₁</td>
<td>d₁</td>
<td>e₁</td>
<td></td>
</tr>
<tr>
<td>c₃</td>
<td>d₂</td>
<td>e₂</td>
<td></td>
</tr>
</tbody>
</table>

Result

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td>d₁</td>
<td>e₁</td>
<td></td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
<td>c₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c₃</td>
<td>d₂</td>
<td>e₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>c₃</td>
<td>d₂</td>
<td>e₂</td>
</tr>
</tbody>
</table>
JOIN VARIANTS

• (full) outer join

Left Table (L):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
<td>c₂</td>
</tr>
</tbody>
</table>

Right Table (R):

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₁</td>
<td>d₁</td>
<td>e₁</td>
</tr>
<tr>
<td>c₃</td>
<td>d₂</td>
<td>e₂</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td>d₁</td>
<td>e₁</td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
<td>c₂</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>c₃</td>
<td>d₂</td>
<td>e₂</td>
</tr>
</tbody>
</table>

• left semi join

Left Table (L):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
<td>c₂</td>
</tr>
</tbody>
</table>

Right Table (R):

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₁</td>
<td>d₁</td>
<td>e₁</td>
</tr>
<tr>
<td>c₃</td>
<td>d₂</td>
<td>e₂</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
</tr>
</tbody>
</table>
JOIN VARIANTS

- **right semi join**

<table>
<thead>
<tr>
<th>L</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td></td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
<td>c₂</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>c₁</td>
<td>d₁</td>
<td>e₁</td>
<td></td>
</tr>
<tr>
<td>c₃</td>
<td>d₂</td>
<td>e₂</td>
<td></td>
</tr>
</tbody>
</table>

= 

<table>
<thead>
<tr>
<th>Resultat</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>c₁</td>
<td>d₁</td>
<td>e₁</td>
<td></td>
</tr>
</tbody>
</table>
Renaming of relation names

- Needed to process self-joins and recursive relationships
- E.g., two-level dependencies of lectures ("grandparents"")

\[ \Pi_{L1.\text{Prerequisite}}(\sigma_{L2.\text{course-id}=CS2270 \land L2.\text{prerequisite}=L1.\text{course-id}}(\rho_{L1}(\text{Requires}) \times \rho_{L2}(\text{Requires}))) \]

Renaming of attribute names

\[ \rho_{\text{Requirement} \leftarrow \text{Prerequisite}}(\text{requires}) \]

<table>
<thead>
<tr>
<th>Requires</th>
<th>course-id</th>
<th>prerequisite</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CS1951A</td>
<td>CS160</td>
</tr>
<tr>
<td></td>
<td>CS1951A</td>
<td>CS320</td>
</tr>
<tr>
<td></td>
<td>CS2270</td>
<td>CS1270</td>
</tr>
<tr>
<td></td>
<td>CS1270</td>
<td>CS160</td>
</tr>
</tbody>
</table>
SET DIFFERENCE (−)

Notation: Relation$_1$ - Relation$_2$

R - S valid only if:

1. R, S have same number of columns (arity)
2. R, S corresponding columns have same domain (compatibility)

Example:

\[(\Pi_{\text{bname}} (\sigma_{\text{amount} \geq 1000} (\text{loan}))) - (\Pi_{\text{bname}} (\sigma_{\text{balance} < 800} (\text{account}))))\]

<table>
<thead>
<tr>
<th>bname</th>
<th>lno</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>L-17</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-23</td>
<td>2000</td>
</tr>
<tr>
<td>Perry</td>
<td>L-15</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>L-14</td>
<td>500</td>
</tr>
<tr>
<td>Perry</td>
<td>L-16</td>
<td>300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mianus</td>
<td>A-215</td>
<td>700</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>Redwood</td>
<td>A-222</td>
<td>700</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>850</td>
</tr>
</tbody>
</table>

Result?

(1) Mianus

(2) Downtown

(3) Redwood

Perry
SET DIFFERENCE (−)

Notation: $\text{Relation}_1 - \text{Relation}_2$

R - S valid only if:

1. $R, S$ have same number of columns (arity)
2. $R, S$ corresponding columns have same domain (compatibility)

Example:

$$(\Pi_{\text{bname}} (\sigma_{\text{amount} \geq 1000} (\text{loan}))) - (\Pi_{\text{bname}} (\sigma_{\text{balance} < 800} (\text{account})))$$

Result?

<table>
<thead>
<tr>
<th>bname</th>
<th>lno</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>L-17</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-23</td>
<td>2000</td>
</tr>
<tr>
<td>Perry</td>
<td>L-15</td>
<td>1500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mianus</td>
<td>A-215</td>
<td>700</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>Redwood</td>
<td>A-222</td>
<td>700</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>850</td>
</tr>
</tbody>
</table>

$$= (1)$$

$$= (2)$$

<table>
<thead>
<tr>
<th>bname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mianus</td>
</tr>
<tr>
<td>Redwood</td>
</tr>
</tbody>
</table>

$$= (3)$$

<table>
<thead>
<tr>
<th>bname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
</tr>
<tr>
<td>Redwood</td>
</tr>
<tr>
<td>Perry</td>
</tr>
</tbody>
</table>
INTERSECTION

\[ \Pi_{\text{Person-ID}}(\text{Lecture}) \cap \Pi_{\text{Person-ID}}(\sigma_{\text{Level}=\text{FP}}(\text{Professor})) \]

Only works if both relations have the same schema

- Same attribute names and attribute domains

Intersection can be simulated with minus:

\[ R \cap S = R - (R - S) \]

Union works similarly...


Codd's Theorem

3 Languages:

• Relational Algebra
• Tuple Relational Calculus (safe expressions only)
• Domain Relational Calculus (safe expressions only)

are equivalent.

Impact of Codd's theorem:

• SQL is based on the relational calculus
• SQL implementation is based on relational algebra
• Codd's theorem shows that SQL implementation is correct and complete.
NOT COVERED

Set Division

Aggregate Functions

Codd’s Proof

...
IN CLASS TASK

<table>
<thead>
<tr>
<th>PlayerID</th>
<th>Name</th>
<th>Age</th>
<th>Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Russel</td>
<td>27</td>
<td>Seahawks</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Team</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seahawks</td>
<td>Washington</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Played</th>
</tr>
</thead>
<tbody>
<tr>
<td>PlayerID</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

In relational algebra:
1) Return all teams, who played at least once in Phoenix
2) Return all Seahawks player, who did not play in the entire season
Return all teams, who played at least once in Phoenix

A) \( \Pi_{\text{Team}} (\sigma_{\text{Place}=`\text{Phoenix}`} (\text{Player} \times \text{Played})) \)

B) \( \Pi_{\text{Team}} \text{ Player } \bowtie (\sigma_{\text{Place}=`\text{Phoenix}`} (\text{Played})) \)

C) \( \Pi_{\text{Team}} (\sigma_{\text{Place}=`\text{Phoenix}`} (\text{Player} \bowtriangle \text{Played})) \)

http://clicker.csail.mit.edu/6.814/
Which of these expressions are equivalent?

A) All
B) 1 and 2
C) 1 and 3
D) 2 and 3
E) None

http://clicker.csail.mit.edu/6.814/
Return all Seahawks player names, who did not play so far

A) $\Pi_{\text{Name}} \left( \sigma_{\text{Team}=\text{Seahawks}} \left( \text{Player} \not\supseteq \text{Played} \right) \right)$

B) $\Pi_{\text{Name}} \left( \sigma_{\text{Team}=\text{Seahawks}} \left( \text{Player} \right) \right) - \Pi_{\text{Name}} \left( \sigma_{\text{Team}=\text{Seahawks}} \left( \text{Player} \nsubseteq \text{Played} \right) \right)$

C) $\Pi_{\text{Name}} \left( \sigma_{\text{Team}=\text{Seahawks}} \left( \text{Player} \nsubseteq \text{Played} \right) \right)$

D) $\Pi_{\text{Name}} \left( \sigma_{\text{Team}=\text{Seahawks} \land \text{Date} = \text{null}} \left( \text{Player} \nsubseteq \text{Played} \right) \right)$
OTHER DATA MODELS

- Object-Oriented Database Systems (OODB)
- Object-Relational Database Systems (ORDB)
- Document stores
- XML Database Systems / Xquery
- RDF
- Key/Value Stores