6.5830 Lecture 2
Sam Madden

Key ideas:
Relational vs Other Data Models
Advanced SQL

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

• **Data models + history**
  – Hierarchical (IMS/DL1) – 1960’s
  – Network (CODASYL) – 1970’s
  – Relational – 1970’s and beyond

• **Key ideas**
  – Data redundancy (and how to avoid it)
  – Physical and logical data independence
  – Relational algebra and axioms
Recap: Zoo Data Model

Entity Relationship Diagram

- Animal
  - name
  - age
  - species
  - entity

- Cage
  - feedTime
  - bldg
  - n contains 1 relationship

- Keeper
  - name
  - n keeps

- Name
  - 1

- Building
  - 1

- Time
  - 1

Animals have names, ages, species
Keepers have names
Cages have cleaning times, buildings
Animals are in 1 cage; cages have multiple animals
Keepers keep multiple cages, cages kept by multiple keepers
Zoo Tables (aka Relations)

## Animals

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>species</th>
<th>cageno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sam</td>
<td>3</td>
<td>Salamander</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>12</td>
<td>Giraffe</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Sally</td>
<td>1</td>
<td>Student</td>
<td>2</td>
</tr>
</tbody>
</table>

## Cages

<table>
<thead>
<tr>
<th>no</th>
<th>feedtime</th>
<th>building</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12:30</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1:30</td>
<td>2</td>
</tr>
</tbody>
</table>

## Keepers

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jane</td>
</tr>
<tr>
<td>2</td>
<td>Joe</td>
</tr>
</tbody>
</table>

## Keeps

<table>
<thead>
<tr>
<th>kid</th>
<th>cageno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

“Schema”: Field names & types
Rows, records, or tuples
Cages in Building 32

• Imperative
  
  for each row a in animals
  for each row c in cages
    if a.cageno = c.no and c.bldg = 32
    output a

• Declarative
  
  SELECT a.name FROM animals AS a, cages AS c
  WHERE a.cageno = c.no AND c.bldg = 32

  Equivalently
  
  SELECT a.name
  FROM animals AS a JOIN cages AS c ON a.cageno = c.no
  WHERE c.bldg = 32
Modified Zoo Data Model

- **animals**: (name, species, age, feed time)
  - livesin: cages (no, size, bldg)
  - caredforby: 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
IMS (Hierarchical Model)

• Data organized as *segments*
  – Collection of records, each with same *segment type*
  – Arranged in a tree of segment types, e.g.:

```
  Keepers
    Animals
    Cages
  Cages
    Animals
```

• Segments have different physical representations
  – Unordered
  – Indexed
    • Sorted
    • Hashed
Example Hierarchy

Jane (keeper) (HSK 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)

Repeated information!

IMS Physical Represenation

Keepers segment

A1 Segment  A2 Segment  A3 Segment
C1 Segment  C2 Segment  C3 Segment
Segment Structure

• Each segment has a particular physical representation
  – Chosen by database administrator
  – E.g., ordered, hashed, unordered…

• Choice of segment structure affects which operations can be applied on it
IMS / DL/1 Operations

• **GetUnique** (seg type, pred)
  – Get first record satisfying pred
  – Only supported by hash / sorted segments

• **GetNext** (seg type, pred)
  – Get first or next key in hierarchical order
  – Starts from last GetNext/GetUnique call

• **GetNextParent** (seg type, pred)
  – Same as GetNext, but will not move up hierarchy to next parent

• **Delete, Insert**
Example PL/1 Program #1

Find the cages that Jane keeps
   GetUnique(Keepers, name = "Jane")
Until done:
   cageid = GetNextParent (cages).no
   print cageid
Example PL/1 Program #2

Find the keepers that keep cage 6
keep = GetUnique(keepers)

Until done:
cage = GetNextParent(cages, id = 6)
if (cage is not null):
    print keep
keep = GetNext(keepers)
What’s Bad About IMS/PL1?

• Duplication of data w/ non-hierarchical data
• Painful low level programming interface – have to program the search algorithm
• Limited **physical data independence**
  – Change root from indexed to hash --- programs that do GN on the root segment will fail
  – Change root from keepers to animals? Also fails.
  – Cannot do inserts into sequential root structure

• Limited **logical data independence**
  – Schemas change, do programs have to?
Logical Data Independence

• Suppose as a cost cutting measure, Zoo management decides a keeper will be responsible for a cage – and all the animals in that cage.

All programs have to change, because the position in the database after a GN/GNP call may not be the same anymore!

Will see how SQL addresses this
Schemas Change for Many Reasons

- Management decides to have “patrons” who buy cages
  - Need to add a patronid column
- Feds change the rules (OSHA)
  - Keepers can keep at most 2 cages
- Tax rules change (IRS)
- Merge with another zoo
Study break #1

• Consider a course schema with students, classes, rooms (each has a number of attributes)

Classes in exactly one room
Students in zero or more classes
Classes taken by zero or more students
Rooms host zero or more classes
Questions

1. Describe one possible hierarchical schema for this data
2. Is there a hierarchical representation that is free of redundancy?
Solution

• Many are possible; one example:
  – Classes
    • Students
      – Rooms
• Duplicates data about students,
  – Students take multiple classes, rooms host multiple classes
• Any other arrangement also duplicates data
CODASYL

- Conference/Committee on Data Systems Languages
  - Responsible for COBOL

- CODASYL data model developed by consortium of large companies in the 70’s

- Designed to address limitations of IMS/PL1

- Graph or network-based data model
Example CODASYL Network

Animals (name, species, age)

Cages (no, size, bldg)

Keepers (name, address)

Records can either be hashes (allowing equality lookup) or sorted ("clustered") according to some key (allowing a range lookup).

Animals: Sam, salamander, Mike, giraffe, Sally, student

Cages: 1, 2

Keepers: Joe, Jane
Example: Find Cages Joe Keeps

Find keepers (name = 'Joe')

Until done:
  Find next animal in caredforby
  Find cage in livesin

• Programming is finding an entry point and navigating around in multidimensional space
  – Each line of code is implicitly at some location in this structure
  – Have to remember where you are
Codasyl Problems

• Incredibly complex — “Navigational Programming”

• Programs lack 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

• Some of this could have been fixed by adding a high-level language to CODASYL

• Relational model was a clean-slate approach designed to fix this
Relational Principles

• Simple representation
• Set-oriented programming model that doesn't require "navigation"
• No physical data model description required(!)
Relational Data Model

• All data is represented as tables of records (*tuples*)
• Tables are unordered sets (no duplicates)
• Database is one or more tables
• Each relation has a *schema* that describes the types of the columns/fields
• Each field is a primitive type -- not a set or relation
• Physical representation/layout of data is not specified (no index types, nestings, etc)
# Zoo Tables

## Animals

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>species</th>
<th>cageno</th>
<th>keptby</th>
<th>feedtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sam</td>
<td>3</td>
<td>Salamander</td>
<td>1</td>
<td>1</td>
<td>10:00 am</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>12</td>
<td>Giraffe</td>
<td>1</td>
<td>2</td>
<td>11:00 am</td>
</tr>
<tr>
<td>3</td>
<td>Sally</td>
<td>1</td>
<td>Student</td>
<td>2</td>
<td>1</td>
<td>1:00 pm</td>
</tr>
</tbody>
</table>

## Cages

<table>
<thead>
<tr>
<th>no</th>
<th>building</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

## Keepers

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jane</td>
</tr>
<tr>
<td>2</td>
<td>Joe</td>
</tr>
</tbody>
</table>

**Schema:** Animals

(id: int,
 name: string,
 age: int,
 species: string,
 cageno: int references cages.no,
 keptby: int references keepers.id,
 feedtime: time )
### Zoo Tables (last lecture)

#### Animals

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>species</th>
<th>cageno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sam</td>
<td>3</td>
<td>Salamander</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>12</td>
<td>Giraffe</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Sally</td>
<td>1</td>
<td>Student</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Cages

<table>
<thead>
<tr>
<th>no</th>
<th>feedtime</th>
<th>building</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12:30</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1:30</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Keepers

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jane</td>
</tr>
<tr>
<td>2</td>
<td>Joe</td>
</tr>
</tbody>
</table>

#### Keeps

<table>
<thead>
<tr>
<th>kid</th>
<th>cageno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Relational Algebra

- **Projection** \((\pi(T,c_1, ..., c_n))\)
  - select a subset of columns \(c_1 .. c_n\)
- **Selection** \((\sigma(T, \text{pred}))\)
  - select a subset of rows that satisfy \(\text{pred}\)
- **Cross Product** \((T_1 \times T_2)\)
  - combine two tables
- **Join** \((\bowtie(T_1, T_2, \text{pred})) = \sigma(T_1 \times T_2, \text{pred})\)
  - combine two tables with a predicate
- Plus set operations (UNION, DIFFERENCE, etc)
- “Algebra” – Closed under its own operations
  - Every expression over relations produces a relation
Join as Cross Product

<table>
<thead>
<tr>
<th>name</th>
<th>cageno</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sam</td>
<td>1</td>
</tr>
<tr>
<td>Mike</td>
<td>1</td>
</tr>
<tr>
<td>Sally</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>no</th>
<th>bldg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
</tr>
</tbody>
</table>

Find animals in bldg. 32

\[
\sigma (\n\bowtie (\n\text{animals},\n\text{cages},\n\text{animals.cageno} = \text{cages.no}\n),\nbldg = 32\n)\n\]

Real implementations do not ever materialize the cross product

<table>
<thead>
<tr>
<th>cageno</th>
<th>no</th>
<th>name</th>
<th>bldg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Sam</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Sam</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Mike</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Mike</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Sally</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Sally</td>
<td>36</td>
</tr>
</tbody>
</table>
Join as Cross Product

<table>
<thead>
<tr>
<th>Animals</th>
<th>Cages</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>cageno</td>
</tr>
<tr>
<td>Sam</td>
<td>1</td>
</tr>
<tr>
<td>Mike</td>
<td>1</td>
</tr>
<tr>
<td>Sally</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>bldg</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
</tr>
</tbody>
</table>

Find animals in bldg. 32

\[ \sigma (\bowtie (\sigma_{\text{bldg} = 32} (\text{animals} \bowtie \text{cages}, \text{animals.cageno} = \text{cages.no}))) \]
### Join as Cross Product

#### Animals

<table>
<thead>
<tr>
<th>name</th>
<th>cageno</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sam</td>
<td>1</td>
</tr>
<tr>
<td>Mike</td>
<td>1</td>
</tr>
<tr>
<td>Sally</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Cages

<table>
<thead>
<tr>
<th>no</th>
<th>bldg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
</tr>
</tbody>
</table>

#### Find animals in bldg. 32

\[
\sigma (\bigtriangleup (\text{animals}, \text{cages}, \text{animals.cageno} = \text{cages.no}), \ bldg = 32)
\]

1. \(\text{animals.cageno} = \text{cages.no}\)
2. \(\ bldg = 32\)

*Do you think this is how databases actually execute joins?*
Relational Identities

• Join reordering
  – $A \bowtie B = B \bowtie A$
  – $(A \bowtie B) \text{ join } C = A \bowtie (B \bowtie C)$

• Selection reordering
  – $\sigma_1(\sigma_2(A)) = \sigma_2(\sigma_1(A))$

• Selection push down
  – $\sigma(A \bowtie_{\text{pred}} B) = \sigma(A) \bowtie_{\text{pred}} \sigma(b)$
  – $\sigma$ may only apply to one table

• Projection push down
  – $\pi(\sigma(A)) = \sigma(\pi(A))$
  – As long as $\pi$ doesn’t remove fields used in $\sigma$
  – Also applies to joins
Push Down Example

\[\sigma (\bowtie (\bowtie (\text{animals, cages, animals.cageno = cages.no}, \text{bldg = 32} ) ) ) ) \]

\[\bowtie (\text{animals, } \sigma (\text{cages, bldg = 32} ) ) ) \]

\[\text{animals.cageno = cages.no} \]
Join Ordering Example

- Find buildings Joe keeps
- SQL

```sql
SELECT building
FROM cages JOIN keeps ON no = cageno
JOIN keepers on kid = id
WHERE name = 'Joe'
```

Best ordering depends on sizes of tables

Filtered keepers may be much smaller

`SQL query executor free to choose either ordering!
Text of SQL query is not an ordering`
Schema:

classes: (cid, c_name, c_rid, ...)
rooms: (rid, bldg, ...)
students: (sid, s_name, ...)
takes: (t_sid, t_cid)

SELECT s_name FROM student,takes,classes
WHERE t_sid=sid AND t_cid=cid
AND c_name='6.830'
Questions

• Write an equivalent relational algebra expression for this query
• Are there other possible expressions?
• Do you think one would be more “efficient” to execute? Why?

```
SELECT s_name FROM student,takes,classes
WHERE t_sid=sid AND t_cid=cid
AND c_name=‘6.830’
```
Solution

SELECT s_name FROM student, takes, classes
WHERE t_sid = sid AND t_cid = cid
AND c_name = '6.830'

Filtering first is probably a good idea

Filtered table is small, so do join with it and classes first

Will formalize this intuition in a few classes
## IMS v CODASYL v Relational

<table>
<thead>
<tr>
<th></th>
<th>IMS</th>
<th>CODASYL</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many to many relationships without redundancy</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Declarative, non “navigational” programming</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>IMS</td>
<td>CODASYL</td>
<td>Relational</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----</td>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>Many to many relationships without redundancy</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Declarative, non “navigational” programming</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Physical data independence</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Physical Independence

Can change representation of data without needing to change code

Example:

```sql
SELECT a.name FROM animals AS a, cages AS c
WHERE a.cageno = c.no AND c.bldg = 32
```

- Nothing about how animals or cages tables are represented is evident
  - Could be sorted, stored in a hash table / tree, etc
  - Changing physical representation will not change SQL
- No specification of implementation
- Both CODASYL and IMS expose representation-dependent operations in their query API
## IMS v CODASYL v Relational

<table>
<thead>
<tr>
<th></th>
<th>IMS</th>
<th>CODASYL</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many to many relationships without redundancy</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Declarative, non “navigational” programming</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>Physical data independence</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>Logical data independence</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
</tbody>
</table>
Logical Data Independence

• What if I want to change the schema without changing the code?
• No problem if just adding a column or table
• Views allow us to map old schema to new schema, so old programs work
  – Even when changing existing fields
Key Idea: View

• View is a logical definition of a table in terms of other tables

• E.g., a view computing animals per cage

CREATE VIEW cage_count as
(SELECT cageno, count(*)
FROM animals JOIN cages ON cageno=no
GROUP by cageno
)

This view can be used just like a table in other queries
Views Example

• Suppose I want to add multiple feedtimes?
• How to support old programs?
  – Rename existing animals table to animals2
  – Create feedtimes table
  – Copy feedtime data from animals2
  – Remove feedtime column from animals2
  – Create a view called animals that is a query over animals2 and feedtimes

```
CREATE VIEW animals as ( SELECT name, age, species, cageno,
  (SELECT feedtime FROM feedtimes WHERE animalid = id LIMIT 1)
  FROM animals2
)
```
### Summary: IMS v CODASYL v Relational

<table>
<thead>
<tr>
<th></th>
<th>IMS</th>
<th>CODASYL</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many to many relationships without redundancy</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Declarative, non “navigational” programming</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>Physical data independence</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>Logical data independence</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
</tbody>
</table>

Next time: Fancy SQL