Relational Model

"Those who cannot remember the past are doomed to repeat it"

Annoncements

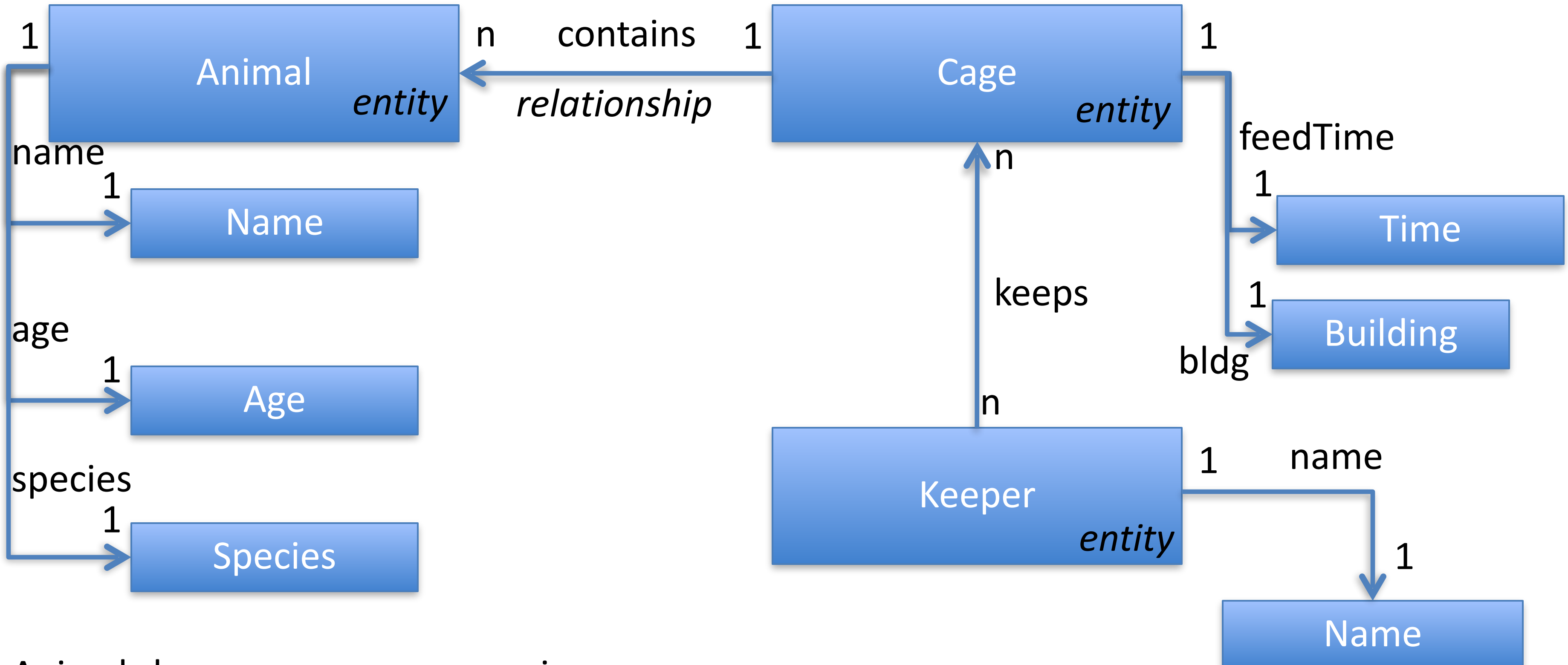
- **PS1 is out. Due next week Wednesday 9/18**
- **Lab1 will be released this Wednesday**
  - First lab requires to get used to system development
  - You must **(pre-)submit a first version of your Lab1 solution by next Thursday 9/19**
  - All pre-submissions, which pass at least 75% of the test cases will get a cupcake the following week during class
  - On 9/20 we will do a lab bootcamp (location and exact time will be announcement this week)
  - You have until 9/25 to make improvements and submit the final solution

A Short History Lesson

- **Different Data Models**
  - Hierarchical (IMS/DL1) – 19*60'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



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

id	name	age	species	cageno
1	Mike	3	Moose	1
2	Tim	12	Giraffe	1
3	Sally	1	Student	2

"Schema": Field names & types

Rows, records, or tuples

Cages

no	feedtime	building
1	12:30	1
2	1:30	2

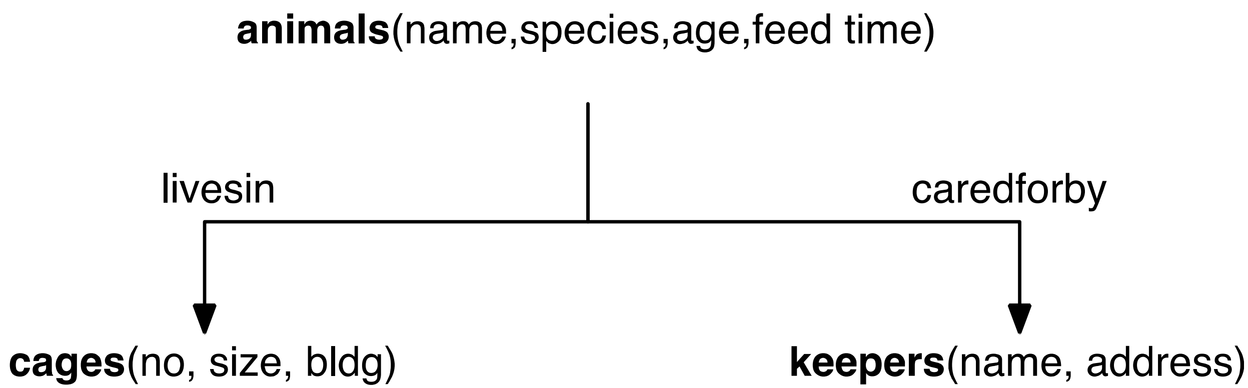
Keepers

id	name
1	Jane
2	Joe

Keeps

kid	cageno
1	1
1	2
2	1

Modified Zoo Data Model



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                 Keepers
    Animals                 Cages
        Cages                   Animals
```

- Segments have different physical representations
  - Unordered
  - Indexed
    - Sorted
    - Hashed

Example Hierarchy

Jane (keeper) (HSK 1)
    Mike, moose, … (2)
        1, 100sq ft, … (3)
    Tim, 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)
  - Get the next segment within the same parent segment
- **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
```

Jane (keeper) (HSK 1)

Mike, moose, … (2)

1, 100sq ft, … (3)

Tim, giraffe, … (4)

2, 1000sq ft, … (5)

Sally, student, … (6)

1, 100sq ft, … (7)

Joe (keeper) (8)

This iterates through data underneath Jane

Implicitly, now navigating from the Jane record in keepers

https://clicker.mit.edu/6.5830/

Find all keepers that take care of cage 6

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

Option A

```
Until done:
    cage = GetNext(cages, id = 6)
    keep = GetNext(keepers)
    print(keep)
```

Option B

```
Until done:
    keep = GetUnique (keepers)
    cage = GetNext(cages, id = 6)
    print(keep)
```

Option C

```
keep = GetUnique(keepers)
    Until done:
        cage = GetNextParent(cages, id = 6)
        if (cage is not null):
                print keep
        keep = GetNext(keepers)
```

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 GetNext 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

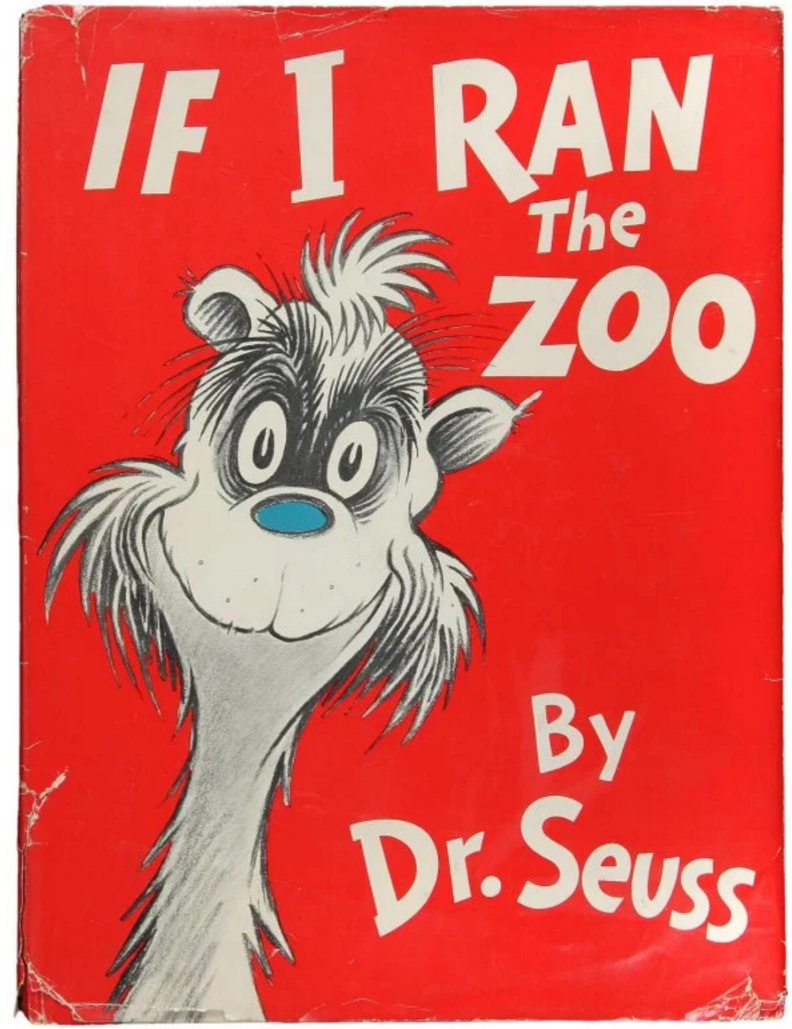
- Suppose as a cost cutting measure, Zoo management decides a keeper will be responsible for a cage – and all the animals in that cage.

Programs have to change, because the position in the database after a GetNext/GetNextParent call may not be the same anymore!

Will see how SQL addresses this

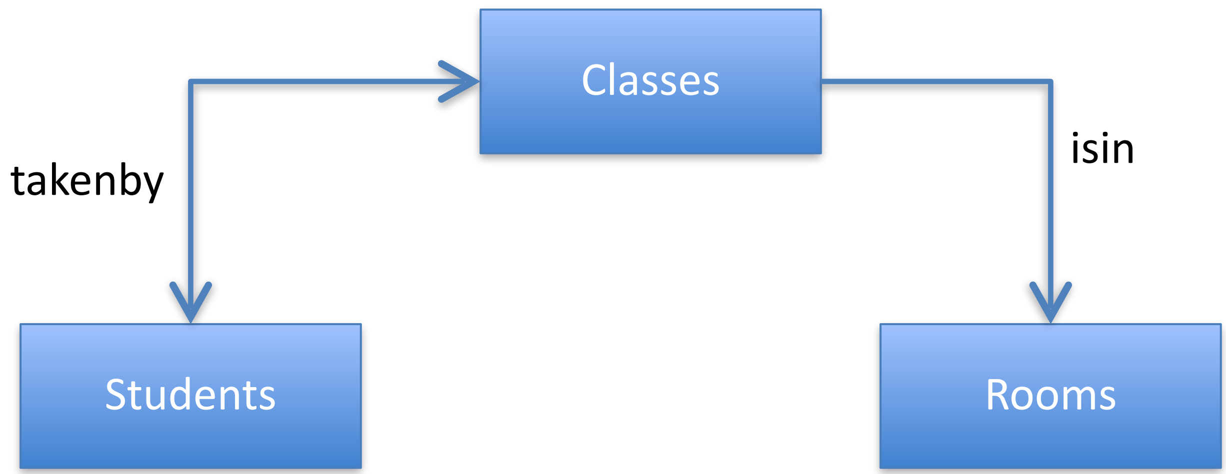
Schemas Change for Many Reasons

- 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 #2

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



Classes in exactly one room

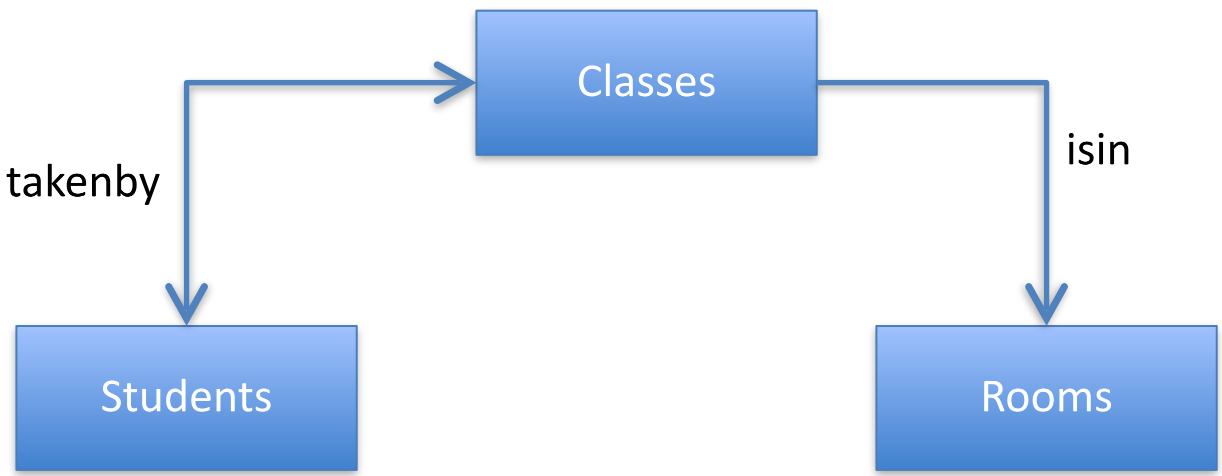
Students in zero or more classes

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 (i.e., the hierarchical model)

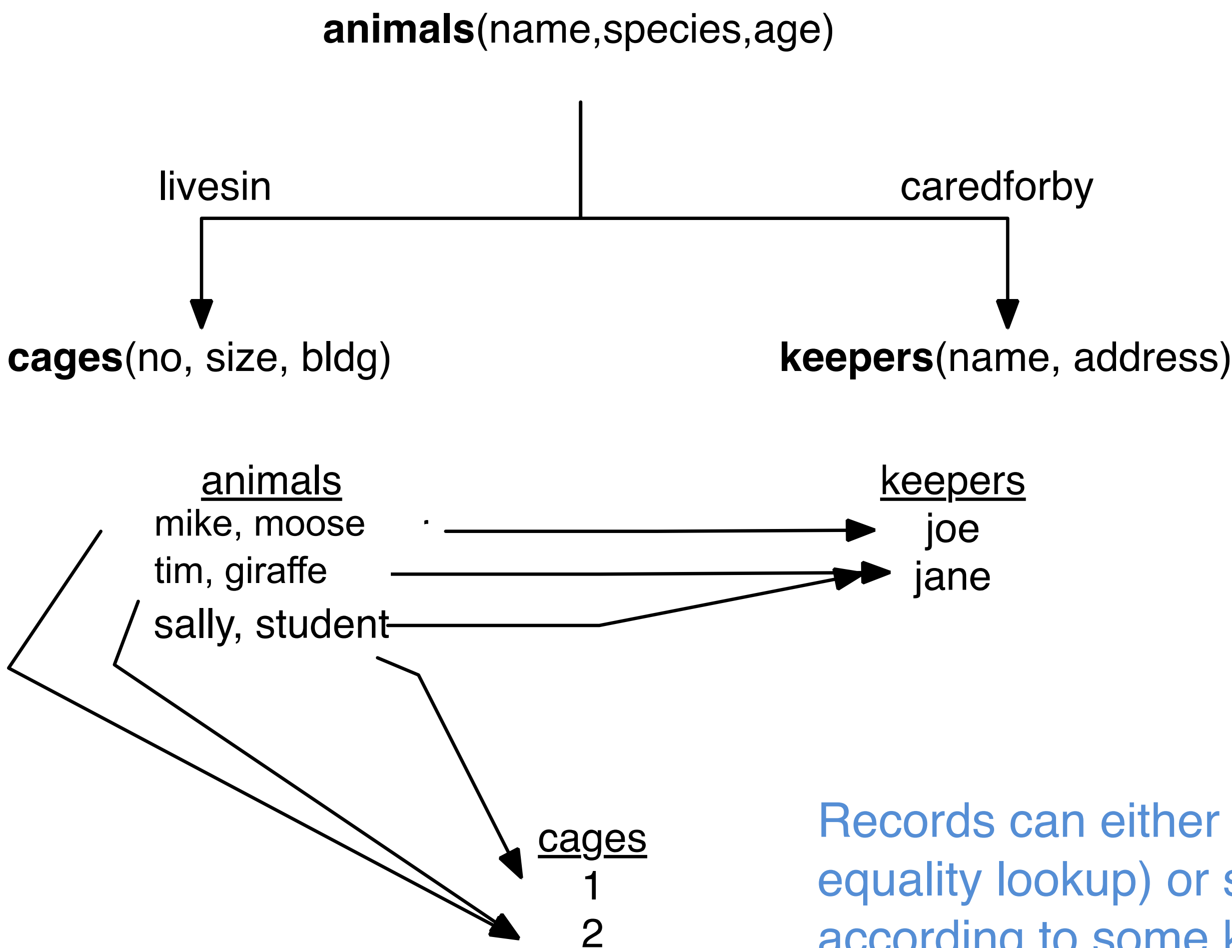
- Graph or network-based data model

The Computer Museum in Boston celebrated COBOL’s 25th anniversary on May 16, 1985. The COBOL tombstone sent to Charles Phillips (see his adjoining article) was presented to the museum. Here surrounding the tombstone (left to right): Ron Hamm, Jack Jones, Jan Prokop, Oliver Smoot, Tom Rice, Donald Nelson, Grace Hopper, Michael O’Connell, and Howard Bromberg (photo by Lilian Kemp). At the museum’s celebration, Bromberg told the following tale of the tombstone.

CODASYL

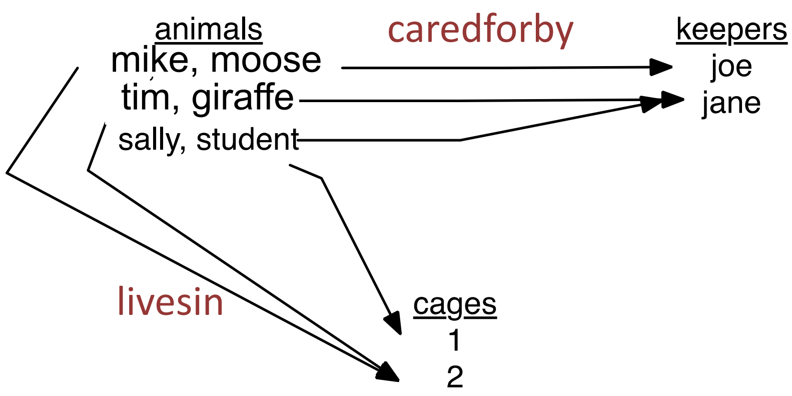
- Networked data model
- Primitives are record types with keys
- Record types are organized into network
- A set consists of one owner record and n member records
- many-to-many links are disallowed, each set occurrence has precisely one owner, and has zero or more member records.
- No member record of a set can participate in more than one occurrence of the set at any point
- **But A member record can participate simultaneously in several set occurrences of *different* sets**.

Example CODASYL Network



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

Example: Find Cages Joe Keeps

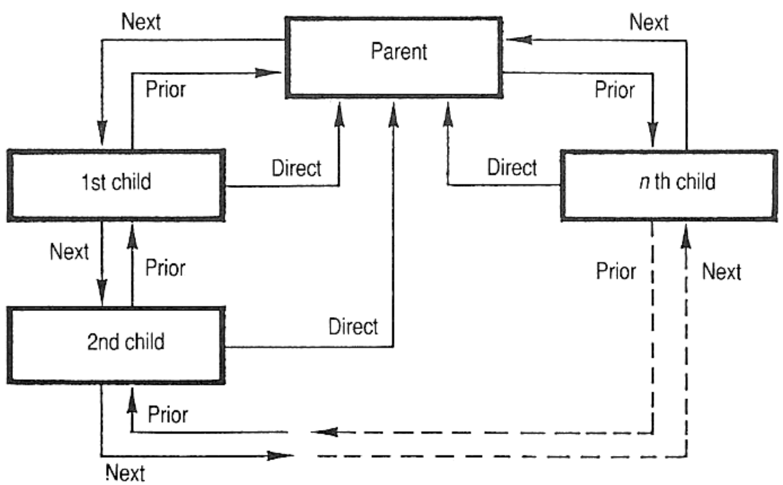


Find keepers (name = 'Joe')

Until done:

Find next animal in caredforb

Find cage in livesin



A closed chain of records in a navigational database model (e.g. CODASYL), with ***next pointers***, ***prior pointers*** and ***direct pointers*** provided by keys in the various records.

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(!)
  - E.g., no specification of sort orders, hashes, etc

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)

*note, modern DBMS often allow duplicates

Zoo Tables

Foreign Keys

Animals

Primary Key

id	name	age	species	cageno	keptby	feedtime
1	Mike	3	Moose	1	1	10:00 am
2	Tim	12	Giraffe	1	2	11:00 am
3	Sally	1	Student	2	1	1:00 pm

Cages

Primary Key

no	building
1	1
2	2

Keepers

Primary Key

id	name
1	Jane
2	Joe

Schema: Animals
(id: int,
name: string,
age: int,
species: string,
cageno: int **references** cages.no,
keptby: int **references** keepers.id.
feedtime: time)

Zoo Tables (original schema)

Animals

id	name	age	species	cageno
1	Mike	3	Moose	1
2	Tim	12	Giraffe	1
3	Sally	1	Student	2

Primary Key

Foreign Key

Cages

no	feedtime	building
1	12:30	1
2	1:30	2

Primary Key

Keepers

id	name
1	Jane
2	Joe

Primary Key

Keeps

kid	cageno
1	1
1	2
2	1

Foreign Key

Foreign Key

Relational Algebra

- **Projection** (π(T,c1, …, cn))
  - select a subset of columns c1 .. cn
- **Selection** (σ(T, pred))
  - select a subset of rows that satisfy pred
- **Cross Product** (T1 x T2)
  - combine two tables
- **Join** ($\bowtie$(T1, T2, pred)) = σ(T1 x T2, 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

cageno	no	name	bldg
1	1	Mike	32
1	2	Mike	36
1	1	Tim	32
1	2	Tim	36
2	1	Sally	32
2	2	Sally	36

Animals

name	cageno
Mike	1
Tim	1
Sally	2

Cages

no	bldg
1	32
2	36

Find animals in bldg. 32

```
σ (
    ⋈(
        animals,
        cages,
        animals.cageno = cages.no
    ),
    bldg = 32
)
```

⇒

σ (
    σ(
        animals X cages,
        animals.cageno = cages.no
    ),
    bldg = 32
)

Join as Cross Product

Animals

name	cageno
Sam	1
Tim	1
Sally	2

Cages

no	bldg
1	32
2	36

cageno	no	name	bldg
1	1	Mike	32
~~1~~	~~2~~	~~Mike~~	~~36~~
1	1	Tim	32
~~1~~	~~2~~	~~Tim~~	~~36~~
~~2~~	~~1~~	~~Sally~~	~~32~~
2	2	Sally	36

Find animals in bldg. 32

σ (
    ⋈(
        animals,
        cages,
        animals.cageno = cages.no
    ),
    bldg = 32
)

σ (
    σ(
        animals X cages,
        animals.cageno = cages.no
    ),
    bldg = 32
)

1. animals.cageno = cages.no

Join as Cross Product

Animals

name	cageno
Sam	1
Tim	1
Sally	2

Cages

no	bldg
1	32
2	36

cageno	no	name	bldg
1	1	Mike	32
~~1~~	~~2~~	~~Mike~~	~~36~~
1	1	Tim	32
~~1~~	~~2~~	~~Tim~~	~~36~~
~~2~~	~~1~~	~~Sally~~	~~32~~
~~2~~	~~2~~	~~Sally~~	~~36~~

1. animals.cageno = cages.no
2. **bldg = 32**

Find animals in bldg. 32

σ (
    ⋈(
        animals,
        cages,
        animals.cageno = cages.no
    ),
    bldg = 32
)

σ (
    σ(
        animals X cages,
        animals.cageno = cages.no
    ),
    bldg = 32
)

Do you think this is how databases actually execute joins?

Relational Identities

- **Join reordering**
  - $A \bowtie B = B \bowtie A$
  - $(A \bowtie B)$ join $C = A \bowtie (B \bowtie C)$

- **Selection reordering**
  - $\sigma_1(\sigma_2(A)) = \sigma_2(\sigma_1(A))$

- **Selection push down**
  - $\sigma(A \bowtie_{pred} B) = \sigma(A) \bowtie_{pred} \sigma(b)$
  - σ may only apply to one table

- **Projection push down**
  - $\pi(\sigma(A)) = \sigma(\pi(A))$
  - As long as π doesn't remove fields used in σ
  - Also applies to joins

Push Down Example

```
σ (
   ⋈ (
     animals,
     cages,
     animals.cageno = cages.no
   ),
   bldg = 32
)
```

```
⋈ (
   animals,
   σ (
        cages,
        bldg = 32
   )
   animals.cageno = cages.no
)
```

Join Ordering Example

- Find buildings Joe keeps
- SQL

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

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

```
⋈ (
    ⋈ (
      cages,
      keeps,
      no = cageno
    ),
    σ (
      keepers,
      name = 'Joe'
    ),
    kid = id
)
```

```
⋈ (
    cages,
    ⋈ (
        σ (
            keepers,
            name = 'Joe'
        ),
      keeps,
      kid=id
    ),
    no = cageno
)
```

Best ordering depends on sizes of tables

Filtered keepers may be *much* smaller

Study Break # 2

Schema:

classes: (<u>cid</u>, c_name, *c_rid*, …)

rooms: (<u>rid</u>, bldg, …)

students: (<u>sid</u>, 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'

- 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?

https://clicker.mit.edu/6.5830/

Schema:

classes: (<u>cid</u>, c_name, *c_rid*, …)

rooms: (<u>rid</u>, bldg, …)

students: (<u>sid</u>, 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'

For the query above select all valid relational algebra expression:

A: π_{s_name} ($\sigma_{c\_name='6.830'}$ (student $\bowtie_{t\_sid=sid}$ (takes $\bowtie_{t\_cid=cid}$ classes)))

B: $\sigma_{c\_name='6.830'}$ (classes $\bowtie_{t\_cid=cid}$ (takes $\bowtie_{t\_sid=sid}$ π_{s_name}(student))

C: π_{s_name} (student $\bowtie_{t\_sid=sid}$ (takes $\bowtie_{t\_cid=cid}$ $\sigma_{c\_name='6.830'}$ (classes)))

D: π_{s_name} ($\sigma_{c\_name='6.830'}$ (classes) $\bowtie_{t\_cid=cid}$ (takes $\bowtie_{t\_sid=sid}$ student))

E: π_{s_name} ($\sigma_{c\_name='6.830'}$ (room $\bowtie_{c\_rid=rid}$ (classes $\bowtie_{t\_sid=sid}$ (takes $\bowtie_{t\_cid=cid}$ student))))

F: π_{s_name} ($\sigma_{c\_name='6.830' \text{ and } t\_sid=sid \text{ and } t\_cid=cid}$ (classes X (takes X student))

https://clicker.mit.edu/6.5830/

Schema:

classes: (<u>cid</u>, c_name, *c_rid*, …)

rooms: (<u>rid</u>, bldg, …)

students: (<u>sid</u>, 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'

Which query plan is most efficient?

A: π_{s_name} $(\sigma_{c\_name='6.830'}$ (student $\bowtie_{t\_sid=sid}$ (takes $\bowtie_{t\_cid=cid}$ classes)))

~~B: $\sigma_{c\_name='6.830'}$ (classes $\bowtie_{t\_cid=cid}$ (takes $\bowtie_{t\_sid=sid}$ π_{s_name}(student))~~

C: π_{s_name} (student $\bowtie_{t\_sid=sid}$ (takes $\bowtie_{t\_cid=cid}$ $\sigma_{c\_name='6.830'}$ (classes)))

D: π_{s_name} $(\sigma_{c\_name='6.830'}$ (classes) $\bowtie_{t\_cid=cid}$ (takes $\bowtie_{t\_sid=sid}$ student))

~~E: π_{s_name} ($\sigma_{c\_name='6.830'}$ (room $\bowtie_{c\_rid=rid}$ (classes $\bowtie_{t\_sid=sid}$ (takes $\bowtie_{t\_cid=cid}$ student))))~~

F: π_{s_name} $(\sigma_{c\_name='6.830' \text{ and } t\_sid=sid \text{ and } t\_cid=cid}$ (classes X (takes X student))