6.5830 Lecture 2 9/12/22

Relational Data Model (PS1 Out)

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

Today we will have a history lesson in data models, and talk about why the relational data model is the way it is.

In the 1970's, a debate raged about data models -- represented on one side by the "academics", who rallied behind Ted Codd and the relational model, and on the other side but the "industrialists" or "pragmatists" who defended the old (network model), and rallied behind Charlie Bachmann.

Culminated in the "great debate" at a SIGMOD workshop in 1974.

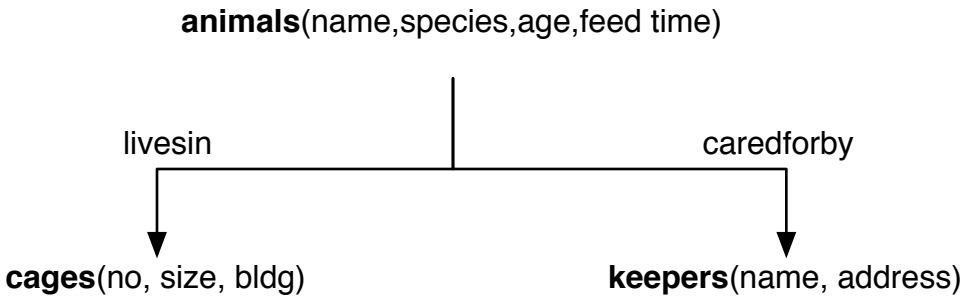
We are going to study 3 models:

1) Hierarchical (IMS + DL/1) — developed in the 1960's for the Apollo program
2) Network (Codasyl + Codasyl DML)
3) Relations (Codd proposal + Relational algebra - SQL / Quel)

Key ideas:

Data redundancy (or how to avoid it)
Physical data independence (programs don't embed knowledge of physical structure)
Logical data independence (logical structure can change and legacy programs can continue to run)
High level languages

Zoo example: (Slide)



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

How would IMS represent this?

IMS is a **hierarchical model**

Uses segment types (record types)
And segments (collections of records of a given type)

Arranged in a tree of segment types

How could we represent this:

Keepers | Animals | Cages	Keepers | Cages | Animals	...

For example for 1)

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)
….

Note that there is repeated information about the cage

Any representation of this schema will repeat information -- introduces the possibility of inconsistency.

There is no redundancy-free representation of this data as a hierarchy! We will see that the programming language doesn't prevent inconsistency from arising.

IMS Physical representation

Keepers segment
|
A1 Segment A2 Segment A3 Segment
C1 Segment C2 Segment C3 Segment

Each segment is a block of records of the same type.

Given a schema, a database administrator (person who sets up the database) chooses 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.

DL/1 is IMS's programming language. Programming in DL/1 is done by explicitly iterating through the database. Basic commands are:

GU (segment type, predicate) - get unique -- optional predicate can be used when root is indexed -- find the first segment satisfying a predicate, and set cursor there
GN (seg, pred) - get the next key, in hierarchical order, moving on to other parents -- optional predicate ; search begins from last GU/GN calls
GNP (seg) - get next record of the specified segment type, stopping when leaving current parent
D - delete this record
I - add a new record

Note that there is an implicit "current position" that the programmer has to keep track of

Find the cages that Jane keeps:

```
GU (Keepers, name = "Jane")
Until done:
        cageid = GNP (cages).no
        print cageid
```

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)
```

<u>IMS problems:</u>

a) duplication of data
b) painful low level programming interface – have to program the search algorithm
c) 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

d) limited logical data independence -- schemas change, would like it if programs didn't have to

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

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

Schemas change for lots of reasons

- Management decides to have “patrons” who buy cages
- Feds change the rules (OSHA)
- Tax rules change (IRS)
- Merge with another zoo

These problems motivated Codd's design.

(break)

Enter Codasyl (Committee on Data Systems Languages)
- COBOL Guys,

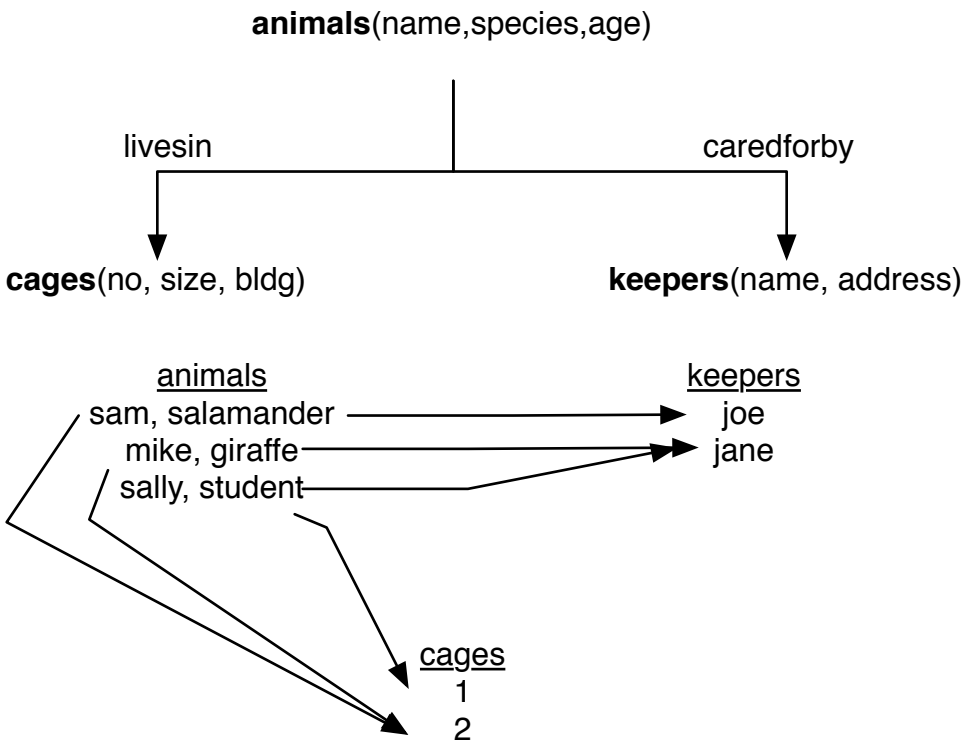
Developed by Charlie Bachman in his IDS system

This was the non-relational camps attempt to address some of the problems with IMS (specifically the fact that it couldn't handle non-hierarchical data)

Approach:

Record types (similar to IMS)
Connected by named sets (1::n relationships)
Arranged in a graph

E.g.: Our example



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

Find the cages that 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 -- have to keep track of where in the set you are at all times, and there are multiple pointers:
    last record read
    last record of each type read
    last record of each set read

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)

Enter the Relational Model

Tedd Codd proposes the relational model as a reaction to the complexity of CODASYL. He was primarily concerned with simplifying the programming model, and providing data independence.

Key properties:
- Simple representation
- Set-oriented programming model that doesn't require "navigation"
- No physical data model description required(!)

The data model:
- All data is represented as tables of records, or 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

"Keys" are used to manage the relationships between records:
- "Primary key" -- unique identifier for a particular record
- "Foreign key" reference to a particular key in another table

Example:
animals

name	age	species	cageno	keptby	feedtime
mike	13	giraffe	1	1	10:00am
sam	3	salam	2	1	11:00am
sally	1	student	1	2	1:00pm

keepers

keeper	name
1	jenny
2	joe

(show keys)

cages

Here name and no are keys, and cageno is a foreign key

Codd proposed the relational algebra -- a very mathy way of expressing operations over tables.

"Algebra" because it is closed -- meaning that the result of every expression is a valid set of relations.

Main operations:

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

Plus various set operations (UNION, DIFFERENCE, etc)

Notice that basic ops are all set oriented, unlike record oriented in previous models (which were "record at a time")

These operations obey interesting algebraic identities that form the basic of query optimization, e.g.

join reordering
A join B = B join A
(A join B) join C = A join (B join C)
sel reordering
Sel1(Sel2(A)) = Sel2(Sel1(A))
sel push down
Sel(A join B, pred) = Sel(A, pred) join Sel(b, pred)
proj push down

(Relational algebra identity examples)

Though the relational algebra is very elegant, it was hard to understand. Part of the great debate was that Bachman et al didn't think mere mortals could program in it, and didn't believe it could be implemented efficiently.

SQL is an attempt to create something more reasonable

instead of math expressions, set-oriented language; above query becomes

```
select cageno
from keepers, animals
where keeper = keptby
```

and keeper.name = 'joe'

(break) (SKIP IF NO TIME)

Let's compare SQL to CODASYL
- **Programming no longer navigates in N-D space** -- just write a program to extract the data they want. Simple, and elegant.

- Physical independence:

Note that the relational model says nothing about the physical representation at all. Users just interact with operators on tables (sets). Tables could be sorted, hashed, indexed, whatever

Programs will continue to run no matter what physical representation is chosen for the data.

What is supposed to happen is that:

- User isn't supposed to know about representation

- Administrator tries to layout data on disk in a way that is good for user's queries. Query optimizer picks best "access method" for query at hand

In practice, as a user of a database, you often need to work to get the best representation to make your queries run fast. But typically this doesn't involve schema changes.

- Logical independence: (How can you change the schema without breaking legacy programs?)

Create a "view" that looks like the old schema.

E.g., suppose we want to add multiple feedtimes

Create a new table, feedtimes (aname string, feedtime time)

Rename the animals table to animals2

Remove the feedtime column from animals

Create a "view" with the name animals, defined in terms of animals2 and feedtimes:

```
CREATE VIEW animals AS
SELECT name, age, species, cageno, keptby,
  (SELECT feedtime
```

```
  FROM feedtimes
  WHERE aname=name
  LIMIT 1)
FROM animals
```

Now when animals2 gets updated, animals will still be valid table according to old schema.

Of course, old apps need to be rewritten if they need to know animals have multiple feed times.

Somewhat imperfect -- e.g., figuring out how to update a2 if animals is updated is hard (this is the "view update problem")

Can also introduce performance impacts, since queries over old schema have to be "rewritten" through views

We will talk more later, but *all* queries and many updates can be supported on views.

In summary -- 1970's there was a long debate on this topic

One one hand, non-relational camp argued for
- Low level efficient programming languages

Relational camp wanted:
- High level languages
- Data independence

Relational camp eventually won -- when IBM released its relational product.

Note that relational isn't the be all and end all of data models. For example, could imagine a data model in which data is *ordered* instead of *unordered*. Would make certain kinds of operations much easier.

For example, imagine a sequence of stock quotes

IBM, 10
IBM, 11
IBM, 10
IBM, 12
IBM, 15
….
Suppose we want to find a sequence of 3 increasing quotes. Relational algebra/SQL: 3

way self join with stocks table -- super inefficient as it requires multiple passes over table.

```
SELECT s1.price,s2.price,s3.price
FROM stocks AS s1, stocks AS s2, stocks AS s3
WHERE s1.sym = s2.sym and s2.sym = s3.sym
AND s2.price > s1.price AND s3.price > s2.price
```

If had sequences, could write an expression like, which would compute the answer in a single pass (enabling "streaming").

```
SELECT stocks.price, stocks.next.price, stocks.next.next.price
FROM stocks
WHERE stocks.price < stocks.next.price AND stocks.next.price < stocks.next.next.price
```

On the other hand, the relational identities don't necessarily hold w/out explicit sorts, complicating execution and optimization of such queries.