Legacy – SQL is a terrible language -- Date paper in 1985

Animals (name, species, age, feeding_time, cid, kid)
Cages (id, size)
Keepers (id, name, address)

Original idea (1974)

Block
    Block
        Block

e.g. find the name of Freddies's keeper

```
select name
from keepers
where id =
    select kid
    from Animals
    where name = "Freddie"
```

Evaluation from inside–out

First: this may not be the best plan

Second: not powerful enough

Find animals older than Freddie who share the same cage

```
Select name
From animals A
Where age >
    Select age
    From animals
    Where name = "Freddie"
    And cid = A.cid
```

Find all pairs of animals cared for by the same keeper

```
Select A.name, B.name
From Animals A
Where a.zid =
    Select B. zid
```

From Animals B
Where B.name != A.name

Requires refs from inside out and outside in

No obvious query processing strategy → disallowed

Okay for an inner to outer reference, but not the other way around.

SQL solution (1976) -- multi-table blocks

Select A.name, B.name
From Aminals A, Animals B
Where A.zid = B.zid and A.name != B.name

Net result: horrible

2 ways to express most queries, e.g. Freddie's keeper:

(nested)

select name
from keepers
where id in
select kid
from Animals
where name = "Freddie"

(flat)

Select k.name
From Animals A, Keepers K
Where A.kid = K.id and A.name = 'Freddie'

Which one to choose?

1980's:

Hierarchical got you inside-out evaluation
Flat got you a complete optimization → obviously better!

Don't use hierarchical representation!!!!

More recently, SQL engines rewrite hierarchical queries to flat ones, if they can. Big pain for the implementation!!!

There are queries that cannot be flattened.

e.g. ones with an "=" between the inner and out block.

There are ones that cannot be expressed in nested fashion, e.g. "pairs of Animals query" above

Two collections of queries: nested and flat: venn diagram almost identical but not quite.

Awful language design.

Lessons

Never too late to toss everything and start over -- glueing warts is never a good idea
Simple is always good
Language design should be done by PL folks not DB folks – we are no good at it
****

OODB (1980s): persistent C++
Motivation:

Programming language world (C++)

```
struct animals {
        string name;
        string feed_time;
        string species;
        int age;
};

struct keeper_data {
        string name;
        string address;
        animals charges[20];
}; DBobject;
```

data base world:

Keepers (name, address)
Animals (name, feed_time, species, age, k_id)

There is an obvious impedance mismatch.

Query returns a table – not a struct. You have to convert DB return to data structures of client program.

OODBs were focused on removing this impedance mismatch.

Data model: C++ data structures. Declare them to be persistent:

e.g. persistent keeper_data DB [500];

to query data base:

temp = DB[1].keeper_data.name

to do an update:

DB [3].keeper_data.name = “Joseph”

Query language: none – use C++ appropriate for CAD –style apps
Assumes programming language Esperanto (C++ only)
Later adopted a QL (OQL)
Transaction systems very weak – big downside

Never found a sweet spot.
****
Semi-structured data problem

Hobbies of employees

Sam: bicycle (brand, derailer, maximum slope hill, miles/week)

Mike: hike (number of 4000 footers, boot brand, speed)
Bike (builder, frame-size, kind-of-seat)

“semi-structured data”
Not well suited to RM (or any other model we have talked about so far)

XML (hierarchical, self-describing tagged data)

```
<employee>
        <name> Sam </name>
        <hobbies>
                <bike>
                        <brand>  XXX </brand>
                                .
                                .
                <next hobby>
```

XML is anything you want.

Document guys proposed this stuff as simplified SGML.

Adapted by DBMS guys. Always a bad idea to morph something to another purpose.

If you want a structured collection, they proposed XML-schema

<employee: schema>
    <employee:element name = “empname” type = “string”/>
    <employee:complextype name = “hobbies”, type = “hobbiestype”/>

XML representation for structure of the XML document
Most complex thing on the planet…
    Tables (RM)
    Hierarchies (IMS style)
    Refs (Codasyl style)
    Set valued attributes (color = {red, green brown})
    Union types (value can be an X or a Y)

XQuery is one of the query languages (with XPath)

For $X in Employee
Where $x/name = Sam
Return $X/hobbies/bike

Huge language

Relational elephants have added (well behaved subsets) to their engines

getting some traction.

******8
Data base design
****
Animals (name, species, age, feeding_time, cid, kid)
Cages (id, size)
Keepers (id, name, address)

Or

Animals (name, species, age, feeding_time)
Cages (id, size)
Keepers (id, name, address)
Lives_in (aname, cid)
Cared_for_by (aname, kid)

Data base design problem – which one to choose

Two ways to data base design

a) Normalization

b) E-R models

Normalization: start with an initial collection of tables, (for this example)

Animals (name, species, age, cid, cage_size)

Functional dependency: for any two collections of columns, second set is determined by the first set; i.e. it is a function.

Written A -> B

In our example:

(name) is a key. Everything is FD on this key

Plus

cid -> size

Problems:

1) redundancy: cage_size repeated for each animal in the case
2) cannot have an empty cage

Issue:

Cage_size is functionally dependent on cid which in turn is functionally dependent on name. So called transitive dependency.

Solution normaliziation;

Cage (id, cage_size)
Animals (name, species, age, c_id)

1NF (Codd) -- all tables “flat”
2NF (Codd) -- no transitive dependencies
3NF (Codd) -- fully functional on key

BCNF
4NF

5NF
P-J NF
….

Theoreticians had a field day….

Totally worthless

1) mere mortals can't understand FDs
2) have to have an initial set of tables – how to come up with these?

Users are clueless…

Plus, if you start with:

Animals (name, species, age, feeding_time)
Cages (id, size)
Keepers (id, name, address)
Lives_in (aname, cid)
Cared_for_by (aname, kid)

No way to get

Animals (name, species, age, feeding_time, cid, kid)
Cages (id, size)
Keepers (id, name, address)

******
Universal solution:

E-R model:

Entities (things with independent existence)

Keepers
Cages
Animals

Entities have attributes
Entities have a key

Animals have name (key), species, age, …

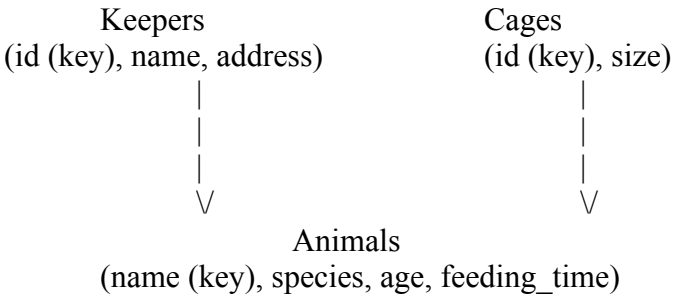
Entities participate in relationships

Lives_in
Cared_for_by

Relationships are 1:1, 1::N or M::N (use crows feet on the arcs to represent visually)

Relationships can have attributes

Draw an E-R diagram



Automatic algorithm generates 3NF (Wong and Katz 1979)

Each entity is a table with the key
M::N relationships are a table with their attributes
1::N relationships – add the key on the N side to the one side with all of the relationship attributes

Generates:

Animals (name, species, age, feeding_time, cid, kid)
Cages (id, size)
Keepers (id, name, address)

Over the years has been extended with

Weak entities (no key – inherits the key of some other entity) (learn to drive)
Inheritance hierarchies (generalization) (student is a specialization of person)
Aggregation (attribute of all of the participants in a relationship) (e.g count)
More than binary relationships (e.g. marriage ceremony)

Details in Ramakrishnan…..

6.830 / 6.814 Database Systems

Fall 2010